ARL-TR-7894 • DEC 2016

US Army Research Laboratory

Vectors and Rotations in 3-Dimensions: Vector Algebra for the C++ Programmer

by Richard Saucier

NOTICES

Disclaimers

ARL-TR-7894 ● DEC 2016

US Army Research Laboratory

Vectors and Rotations in 3-Dimensions: Vector Algebra for the C++ Programmer

by Richard Saucier
Survivability/Lethality Analysis Directorate, ARL

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED (From - To)**
December 2016	Technical Report	January 2016–July 2016

4. TITLE AND SUBTITLE Vectors and Rotations in 3-Dimensions: Vector Algebra for the C++ Programmer	**5a. CONTRACT NUMBER** **5b. GRANT NUMBER** **5c. PROGRAM ELEMENT NUMBER**
6. AUTHOR(S) Richard Saucier	**5d. PROJECT NUMBER** AH80 **5e. TASK NUMBER** **5f. WORK UNIT NUMBER**
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Research Laboratory ATTN: RDRL-SLB-S Aberdeen Proving Ground, MD 21005-5068	**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-7894
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSOR/MONITOR'S ACRONYM(S)** **11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes 2 C++ classes: a Vector class for performing vector algebra in 3-dimensional space (3D) and a Rotation class for performing rotations of vectors in 3D. Each class is self-contained in a single header file (`Vector.h` and `Rotation.h`) so that a C++ programmer only has to include the header file to make use of the code. Examples and reference sheets are provided to serve as guidance in using the classes.

15. SUBJECT TERMS

vector, rotation, 3D, quaternion, C++ tools, rotation sequence, Euler angles, yaw, pitch, roll, orientation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Richard Saucier
Unclassified	Unclassified	Unclassified	UU	98	**19b. TELEPHONE NUMBER (Include area code)** 410-278-6721

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

List of Listings

INTENTIONALLY LEFT BLANK.

1. Introduction

This report describes 2 C++ classes: a Vector class for performing vector algebra in 3-dimensional space (3D) and a Rotation class for performing rotations of vectors in 3D. These classes give the programmer the ability to use vectors and rotation operators in 3D as if they were native types in the C++ language. Thus, the code

```
Vector c = a + b;   // addition of two vectors
```

performs vector addition, accounting for both magnitude and direction of the vectors to satisfy the parallelogram law of vector addition in exactly the same way as the vector algebra expression $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$. We also take advantage of the operator overloading capabilities of C++ so that operations can be written in a more natural style, similar to that of vector algebra. Thus, the code

```
Vector c = a * b;   // dot product of two vectors
```

expresses the scalar dot product $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b}$,

```
Vector c = a ^ b;   // cross product of two vectors
```

expresses the vector cross product $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$, and

```
Vector c = R * a;   // rotation of a vector
```

expresses a rotation of the vector $\boldsymbol{a}$ by the rotation operator R to give a vector $\boldsymbol{c}$. A reference sheet for each class is made available in Appendix A and Appendix B.

Rotations only require an *axis* and an *angle* of rotation—which is how they are stored—and may be specified in a number of convenient ways. We also provide methods for converting from the internal representation to the equivalent quaternion and rotation matrix representation. Quaternion algebra is summarized in Appendix C, which then provides a coordinate-free formula for the rotation of a vector.

It is also useful to describe rotations as a sequence of 3 standard rotations (Euler angles or yaw, pitch, and roll), and Appendix D shows that a rotation sequence about body axes is equivalent to the same rotation sequence applied in reverse order about fixed axes. There are a total of 12 rotation sequences that can be used to describe the orientation of a vector. In Appendix E we provide formulas[1] for factoring an arbitrary rotation into each of these rotation sequences.

Rotations are commonly described with rotation matrices. Appendix F provides formulas and source code for converting between our descriptions of rotations, the quaternion representation, and the rotation matrix.

Quaternions are also very convenient and efficient for describing smooth rotations between 2 different orientations. Appendix G provides a derivation of the spherical linear interpolation (Slerp) formula for this purpose. We also provide a formula and coding for fast incremental Slerp.

Sometimes we need to relate 2 different orientations and find the rotation that will transform from one to the other. This is called the *absolute orientation problem* and Appendix H provides an exact solution to this problem.

The Rotation and Vector classes provide C++ support for all these operations. No libraries are required and there is nothing to build; one merely needs to include the header file to make use of the class. (The Rotation class includes the Vector class, so one only needs to include `Rotation.h` to also make use of the Vector class.)

We also provide examples of how these classes can be used to solve real problems.

2. Vector Class

The source code for the Vector class is completely self-contained in the header file `Vector.h`, which is listed and described in Appendix A. The program in Listing 1 provides some examples of how one might use the Vector class.

Listing 1. vtest.cpp

```cpp
// vtest.cpp: simple program to demonstrate basic usage of Vector class

#include "Vector.h"   // only need to include this header file
#include <iostream>
#include <cstdlib>
using namespace va;   // vector algebra namespace

int main( void ) {

   // let u be a unit vector that has equal components along all 3 axes
   Vector u = normalize( Vector( 1., 1., 1. ) );

   // output the vector
   std::cout << "u = " << u << std::endl;

   // show that the magnitude is 1
   std::cout << "magnitude = " << u.r() << std::endl;

   // output the polar angle in degrees
   std::cout << "polar angle (deg) = " << deg( u.theta() ) << std::endl;

   // output the azimuthal angle in degrees
   std::cout << "azimuthal angle (deg) = " << deg( u.phi() ) << std::endl;

   // output the direction cosines
   std::cout << "direction cosines = " << u.dircos( X ) << " " << u.dircos( Y ) << " " << u.dircos( Z ) << std::endl;

   // let ihat, jhat, khat be unit vectors along x-axis, y-axis, and z-axis, respectively
   Vector ihat( 1., 0., 0. ), jhat( 0., 1., 0. ), khat( 0., 0., 1. );

   // output the vectors
   std::cout << "ihat = " << ihat << std::endl;
   std::cout << "jhat = " << jhat << std::endl;
   std::cout << "khat = " << khat << std::endl;

   // rotate ihat, jhat, khat about u by 120 degrees
   ihat.rotate( u, rad( 120. ) );
   jhat.rotate( u, rad( 120. ) );
   khat.rotate( u, rad( 120. ) );

   // output the rotated vectors
   std::cout << "after 120 deg rotation about u:" << std::endl;
   std::cout << "ihat is now = " << ihat << std::endl;
   std::cout << "jhat is now = " << jhat << std::endl;
   std::cout << "khat is now = " << khat << std::endl;

   // define two vectors, a and b
   Vector a( 2., 1., -1. ), b( 3., -4., 1. );
   std::cout << "a = " << a << std::endl;
   std::cout << "b = " << b << std::endl;
   std::cout << "a + b = " << a + b << std::endl;
   std::cout << "a - b = " << a - b << std::endl;

   // compute and output the dot product
   double s = a * b;
   std::cout << "dot product, a * b = " << s << std::endl;

   // compute and output the cross product
   Vector c = a ^ b;
   std::cout << "cross product, a ^ b = " << c << std::endl;

   // output the angle (deg) between a and b
   std::cout << "angle between a and b (deg) = " << deg( angle( a, b ) ) << std::endl;

   // compute and output the projection of a along b
   std::cout << "proj( a, b ) = " << proj( a, b ) << std::endl;

   // rotate a and b 120 deg about u
   a.rotate( u, rad( 120. ) );
   b.rotate( u, rad( 120. ) );
   std::cout << "after rotating a and b 120 deg about u: " << a << std::endl;
```

```cpp
    std::cout << "a is now = " << a << std::endl;
    std::cout << "b is now = " << b << std::endl;

    // output the angle (deg) between a and b
    std::cout << "angle between a and b (deg) is now = " << deg( angle( a, b ) ) << std::endl;

    // compute and output the dot product
    s = a * b;
    std::cout << "dot product, a * b is now = " << s << std::endl;

    // compute and output the cross product
    c = a ^ b;
    std::cout << "cross product, a ^ b is now = " << c << std::endl;

    // set a and b to their original values and compute the cross product
    a = Vector( 2., 1., -1. );
    b = Vector( 3., -4., 1. );
    c = a ^ b;
    std::cout << "original cross product, c = " << c << std::endl;

    // rotate c 120 deg about u and output
    c.rotate( u, rad( 120. ) );
    std::cout << "now rotate c 120 deg about u:" << std::endl;
    std::cout << "c is now = " << c << std::endl;

    return EXIT_SUCCESS;
}
```

Save this to a file vest.cpp and compile it with the command

```
g++ -O2 -Wall -o vtest vtest.cpp -lm
```

Running it

```
./vtest
```

will print the following:

```
u = 0.57735 0.57735 0.57735
magnitude = 1
polar angle (deg) = 54.7356
azimuthal angle (deg) = 45
direction cosines = 0.57735 0.57735 0.57735
ihat = 1 0 0
jhat = 0 1 0
khat = 0 0 1
after 120 deg rotation about u:
ihat is now = 0 1 0
jhat is now = 0 0 1
khat is now = 1 0 0
a = 2 1 -1
b = 3 -4 1
a + b = 5 -3 0
a - b = -1 5 -2
dot product, a * b = 1
cross product, a ^ b = -3 -5 -11
angle between a and b (deg) = 85.4078
proj( a, b ) = 0.115385 -0.153846 0.0384615
after rotating a and b 120 deg about u: -1 2 1
a is now = -1 2 1
b is now = 1 3 -4
angle between a and b (deg) is now = 85.4078
dot product, a * b is now = 1
cross product, a ^ b is now = -11 -3 -5
original cross product, c = -3 -5 -11
now rotate c 120 deg about u:
c is now = -11 -3 -5
```

3. Rotation Class

Similarly, the source code for the Rotation class is completely self-contained in the header file `Rotation.h`, which is listed and described in Appendix B. The program in Listing 2 provides some basic examples of usage.

Listing 2. rtest.cpp

```
// rtest.cpp

#include "Rotation.h"
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <iomanip>
using namespace va;

int main( void ) {

   // declare two unit vectors, u and v
   Vector u = Vector( 1.5, 2.1, 3.2 ).unit(), v = Vector( 1.2, 3.5, -2.3 ).unit();
   std::cout << "u        = " << u << std::endl;
   std::cout << "v        = " << v << std::endl;

   // let R be the rotation specified by the vector cross product u ^ v
   Rotation R( u, v );

   // show that R * u equals v
   std::cout << "R * u    = " << R * u << std::endl;

   // interpolate a unit vector, w, halfway between u and v
   Vector w = slerp( u, v, 0.5 );
   std::cout << "Vector halfway between u and v = " << w << std::endl;

   // output angle between u and v
   std::cout << "angle between u and v (deg) = " << deg( angle( u, v ) ) << std::endl;

   // output angle between u and w
   std::cout << "angle between u and w (deg) = " << deg( angle( u, w ) ) << std::endl;

   // let Rinv be the inverse rotation
   Rotation Rinv = inverse( R );

   // show that Rinv * v equals u
   std::cout << "Rinv * v = " << Rinv * v << std::endl;

   Vector ihat( 1., 0., 0. ), jhat( 0., 1., 0. ), khat( 0., 0., 1. );
   R = Rotation( ihat, jhat, khat, jhat,khat, ihat );
   std::cout << "R = " << R << std::endl;

   sequence s = factor( R, ZYX );
   std::cout << "yaw (deg )   = " << deg( s.first ) << std::endl;
   std::cout << "pitch (deg ) = " << deg( s.second ) << std::endl;
   std::cout << "roll (deg )  = " << deg( s.third ) << std::endl;

   R = Rotation( s.first, s.second, s.third, ZYX );
   std::cout << "R = " << R << std::endl;

   s = factor( R, XYZ );
   std::cout << "pitch (deg ) = " << deg( s.first ) << std::endl;
   std::cout << "yaw (deg )   = " << deg( s.second ) << std::endl;
   std::cout << "roll (deg )  = " << deg( s.third ) << std::endl;

   std::streamsize ss = std::cout.precision();
   R = Rotation( s.first, s.second, s.third, XYZ );
   std::cout << "R = " << R << std::endl;

   rng::Random rng;
   R = Rotation( rng );
   std::cout << "R = " << R << std::endl;
   quaternion q = to_quaternion( R );
   std::cout << "the quaternion for this rotation is" << std::endl;
   std::cout << "q = " << q << std::endl;
   R = Rotation( q );
   std::cout << "R = " << R << std::endl;
   matrix A = to_matrix( R );
   std::cout << "the matrix for this rotation is" << std::endl;
   std::cout << std::setprecision(6) << std::fixed << std::showpos;
   std::cout << A << std::endl;
```

```

    std::cout << std::setprecision(ss) << std::noshowpos;
    R = Rotation( A );
    std::cout << "R = " << R << std::endl;

    u = normalize( Vector( 1., 1., 1. ) );
    double th = rad( 120. );
    R = Rotation( u, th );
    std::cout << R << std::endl;
    A = to_matrix( A );
    std::cout << A << std::endl;
    q = to_quaternion( R );
    std::cout << q << std::endl;

    return EXIT_SUCCESS;
}
```

Writing this to a file `rtest.cpp`, compiling and running it,

```
g++ -O2 -Wall -o rtest rtest.cpp -lm
./rtest
```

produces the following output:

```
u        = 0.364878 0.510829 0.778407
v        = 0.275444 0.803378 -0.527934
R * u    = 0.275444 0.803378 -0.527934
Vector halfway between u and v = 0.431716 0.886061 0.168873
angle between u and v (deg) = 84.264
angle between u and w (deg) = 42.132
Rinv * v = 0.364878 0.510829 0.778407
R = 0.57735 0.57735 0.57735     120
yaw (deg )   = -90
pitch (deg ) = -180
roll (deg )  = -90
R = 0.57735 0.57735 0.57735     120
pitch (deg ) = 45
yaw (deg )   = 90
roll (deg )  = 45
R = 0.57735 0.57735 0.57735     120
R = 0.338694 0.929155 0.148182  174.087
the quaternion for this rotation is
q = 0.0515815   0.338243 0.927918 0.147985
R = 0.338694 0.929155 0.148182  174.087
the matrix for this rotation is
-0.765862       +0.612457       +0.195837
+0.642990       +0.727384       +0.239741
+0.004383       +0.309530       -0.950880
R = 0.338694 0.929155 0.148182  174.086566
```

4. Example Applications

The preceding programs exercised some basic capabilities of the 2 classes. Now let us consider some more practical problems to see how these classes aid in their solution. Vectors are powerful tools in 3D problems and it is usually better to make use of the vectors directly in vector algebra rather than to decompose into coordinates, and these classes allow us to do that.

4.1 Impact Location on a Target Plate

Here is an example problem. We have a target plate that is initially placed in the x-y plane, as shown in Fig. 1, where L is the length, W the width, and T the thickness of the plate, defined such that $L \geq W \geq T$. We first define a *standard orientation* of the target plate as the initial orientation and then describe the operational procedure to give it a specific, final orientation. This standard orientation is with the center at the origin of a right-handed cartesian (x, y, z) coordinate system and its length along the x-axis, its width along the y-axis, and its thickness along the z-axis, as illustrated in Fig. 1. Once it has been given a final orientation, we translate its center to the location $\boldsymbol{r}_c$. Then we shoot a fragment at the plate along a ray from the origin and we want to find the impact point on the target, along with its distance and its impact obliquity.

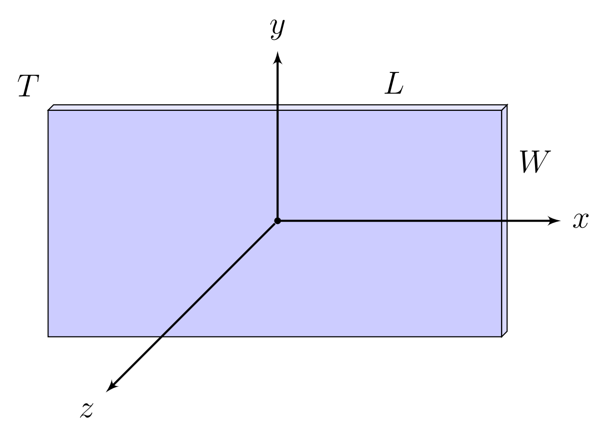

Fig. 1. Initial orientation of the target plate in a right-handed Cartesian coordinate system

We also define the 3 unit vectors $\hat{\mathbf{\imath}}$, $\hat{\mathbf{\jmath}}$, and $\hat{\mathbf{k}}$ along the x-axis, y-axis, and z-axis, respectively. The target plate (entrance) normal $\hat{\boldsymbol{n}}$ is initially aligned with $\hat{\mathbf{k}}$. The final orientation is specified by performing a pitch-yaw-roll rotation sequence, where ϕ_p is pitch about the x-axis, ϕ_y is yaw about the y-axis, and ϕ_r is roll about the

z-axis, as shown in Fig. 2.

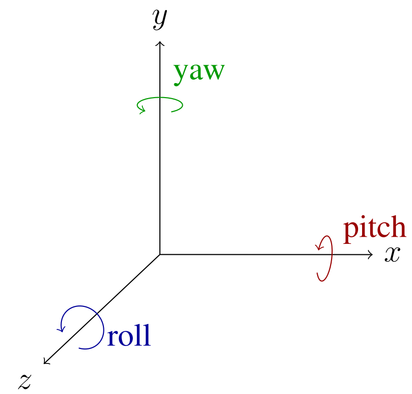

Fig. 2. Description of pitch, yaw, and roll

Here we adopt the FATEPEN (Fast Air Target Encounter Penetration)[2,3] convention that first pitch is applied, then yaw, then roll.* We denote a rotation about the unit axial vector $\hat{\boldsymbol{e}}$ through an angle ϕ with the notation $R_{\hat{\boldsymbol{e}}}(\phi)$. A rotation sequence can thus be written as

$$R \equiv R_{\hat{\mathbf{k}}''}(\phi_r) R_{\hat{\mathbf{j}}'}(\phi_y) R_{\hat{\mathbf{i}}}(\phi_p), \tag{1}$$

where the order is from right to left: first pitch is applied, then yaw, then roll. Notice that we have primes on the yaw and roll rotation operators to indicate that the unit vectors get transformed after pitch is applied and after yaw is applied, since the rotations are applied to the body axes of the plate. Now it is a fundamental theorem of rotation sequences that rotations about the body axes is equivalent to the same sequence in reverse order about the fixed axes†, and since it is more efficient to rotate about fixed axes, we have

$$R = R_{\hat{\mathbf{k}}''}(\phi_r) R_{\hat{\mathbf{j}}'}(\phi_y) R_{\hat{\mathbf{i}}}(\phi_p) = R_{\hat{\mathbf{i}}}(\phi_p) R_{\hat{\mathbf{j}}}(\phi_y) R_{\hat{\mathbf{k}}}(\phi_r). \tag{2}$$

Hence, the normal vector is given by

$$\hat{\boldsymbol{n}} = R_{\hat{\mathbf{i}}}(\phi_p) R_{\hat{\mathbf{j}}}(\phi_y) R_{\hat{\mathbf{k}}}(\phi_r)\, \hat{\mathbf{k}}. \tag{3}$$

Once the target plate is oriented, we specify the location of its center with the vector $\mathbf{r}_c$. So the final position and orientation of the target plate is specified by the pair of vectors $(\mathbf{r}_c, \hat{\boldsymbol{n}})$.

*There are a total of 12 different conventions in the Rotation class that one may choose from.
†See Appendix D for a proof.

Let $\mathbf{r}_0$ be the origin of the fragment ray and let $\hat{\boldsymbol{u}}$ be a unit vector along the ray. We specify the orientation of $\hat{\boldsymbol{u}}$ by specifying the polar angle θ and azimuth angle ϕ, so that

$$\hat{\boldsymbol{u}} = \sin\theta\cos\phi\hat{\mathbf{i}} + \sin\theta\sin\phi\hat{\mathbf{j}} + \cos\theta\hat{\mathbf{k}}, \tag{4}$$

and the equation for the fragment ray is

$$\mathbf{r} = \mathbf{r}_0 + t\hat{\boldsymbol{u}}, \tag{5}$$

where $t \geq 0$ is a scalar that measures the distance from the origin. The target plane (infinite in extent) is specified by all points $\mathbf{r}$ such that

$$(\mathbf{r} - \mathbf{r}_c) \cdot \hat{\boldsymbol{n}} = 0, \tag{6}$$

since the normal, by definition, is orthogonal to the plane. Therefore, the intersection of the fragment ray with the plane of the target is specified by the vector equation

$$(\mathbf{r}_0 + t\hat{\boldsymbol{u}} - \mathbf{r}_c) \cdot \hat{\boldsymbol{n}} = 0. \tag{7}$$

Solving for the distance t gives

$$t = \frac{(\mathbf{r}_c - \mathbf{r}_0) \cdot \hat{\boldsymbol{n}}}{\hat{\boldsymbol{u}} \cdot \hat{\boldsymbol{n}}}. \tag{8}$$

There will be an intersection, provided $\hat{\boldsymbol{u}} \cdot \hat{\boldsymbol{n}} \neq 0$, and the intersection point is

$$\boxed{\mathbf{r} = \mathbf{r}_0 + \frac{(\mathbf{r}_c - \mathbf{r}_0) \cdot \hat{\boldsymbol{n}}}{\hat{\boldsymbol{u}} \cdot \hat{\boldsymbol{n}}}\,\hat{\boldsymbol{u}}}. \tag{9}$$

The impact obliquity is given by

$$\theta_{\text{obl}} = \cos^{-1}(|\hat{\boldsymbol{u}} \cdot \hat{\boldsymbol{n}}|). \tag{10}$$

The location of the impact point with respect to the target center is

$$\mathbf{r} - \mathbf{r}_c = \mathbf{r}_0 - \mathbf{r}_c + \frac{(\mathbf{r}_c - \mathbf{r}_0) \cdot \hat{\boldsymbol{n}}}{\hat{\boldsymbol{u}} \cdot \hat{\boldsymbol{n}}}\,\hat{\boldsymbol{u}} \tag{11}$$

To determine if the intersection point on the infinite target plane lies within the finite target plate, we apply the inverse rotation R^{-1} to the vector $\mathbf{r} - \mathbf{r}_c$ and check to see

if the resulting vector lies within the plate dimensions. Thus, let

$$\mathbf{d} = R^{-1}(\mathbf{r} - \mathbf{r}_c) \tag{12}$$

and then we check to see if

$$-L/2 \le d_x \le L/2 \quad \text{and} \quad -W/2 \le d_y \le W/2. \tag{13}$$

Listing 3 is an implementation of these vector equations.

Listing 3. geometry.cpp

```cpp
// geometry.cpp: compute hit point on an oriented target plate from a general fragment ray

#include "Rotation.h"
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <iomanip>
#include <cassert>
using namespace std;

int main( void ) {

   // target dimensions (in)
   const double L = 96., W = 48., L_2 = L / 2., W_2 = W / 2.;

   // constant unit vectors for the laboratory frame
   const va::Vector I( 1., 0., 0 ), J( 0., 1., 0. ), K( 0., 0., 1. );

   std::cout << std::setprecision(3) << std::fixed;

   va::Vector n = K;                // normal vector is initially along K
   va::Vector u;                    // unit vector along fragment ray
   va::Vector rc( 0., 0., -24. );   // location of the target center
   va::Vector r0( 0., 0., 0. );     // origin of the fragment ray

   // specify pitch, yaw and roll of the target (degrees converted to radians)
   double pitch = va::rad( +30. );   // pitch (converted to rad)
   double yaw   = va::rad( -35. );   // yaw (converted to rad)
   double roll  = va::rad( +45. );   // roll (converted to rad)

   // specify the target orientation with R and inverse with Rinv
   va::Rotation R( pitch, yaw, roll, va::XYZ );
   va::Rotation Rinv = va::inverse( R );

   // apply the orientation to the target normal
   n = R * n;

   double az = -35., el = 15.;   // specify azimuth and elevation of fragment ray (deg)
   assert( -360. <= az && az <= 360. );
   assert( 0. <= el && el <= 90. );
   double th = va::rad( 180. - el );   // convert to polar angle (rad)
   double ph = va::rad( az );          // convert to azimuthal angle (rad)

   // specify direction of u vector
   u = sin( th ) * cos( ph ) * I + sin( th ) * sin( ph ) * J + cos( th ) * K;

   double obl = angle( u, -n );   // obliquity angle (rad)
   if ( obl >= M_PI_2 ) {
      cout << "ray missed target since obl (deg) = " << obl * va::R2D << endl;
      exit( EXIT_SUCCESS );
   }

   // compute distance to (infinite) target plane
   double t = ( rc - r0 ) * n / ( u * n );
   va::Vector r = r0 + t * u;   // hit point in lab frame
   va::Vector d = r - rc;       // hit point relative to the target center
   d = Rinv * d;                // inverse rotation back to the laboratory reference frame

   double xhit = d.x(), yhit = d.y();

   // check if hit point on plane lies with target plate
   if ( ( -L_2 <= xhit && xhit <= L_2 ) && ( -W_2 <= yhit && yhit <= W_2 ) ) {
```

```
        cout << "ray hit target at " << d << " with respect to the target at the origin" << endl;
        cout << "obl (deg) = " << obl * va::R2D << endl;
        cout << "distance  = " << t << endl;
    }
    else {
        cout << "ray missed target plate" << endl;
        cout << "xhit = " << xhit << endl;
        cout << "yhit = " << yhit << endl;
    }

    return EXIT_SUCCESS;
}
```

Compiling and running this program gives the following output:

```
ray hit target at 2.794 -5.560 0.000 with respect to the target at the origin
obl (deg) = 41.752
distance  = 22.822
```

4.2 Computing the Rotation between 2 Orientations

We may know the unit vectors $\hat{\imath}$, $\hat{\jmath}$, $\hat{\mathbf{k}}$ in 2 different reference frames and we need to find the rotation that takes one to the other. Thus, let us suppose that we have 2 sets of orthonormal (basis) vectors, $(\hat{\imath}, \hat{\jmath}, \hat{\mathbf{k}})$ and $(\hat{\imath}', \hat{\jmath}', \hat{\mathbf{k}}')$, and we want to find the rotation R such that

$$\hat{\imath}' = R\,\hat{\imath}, \quad \hat{\jmath}' = R\,\hat{\jmath}, \quad \text{and} \quad \hat{\mathbf{k}}' = R\,\hat{\mathbf{k}}. \tag{14}$$

This is a relatively simple problem for unit vectors. A much more difficult problem is to find the rotation between 2 sets of 3 vectors when the vectors are *not* unit vectors. A closed-form solution to this problem is summarized in Appendix H. Both of these cases have been implemented in the Rotation class. The Listing 4 provides a demonstration of this.

Listing 4. r2.cpp

```
// r2.cpp: find the rotation, given two sets of vectors related by a pure rotation

#include "Rotation.h"
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <iomanip>
#include <cassert>
using namespace std;

int main( int argc, char* argv[] ) {

    // constant unit vectors for the laboratory frame
    const va::Vector i( 1., 0., 0 ), j( 0., 1., 0. ), k( 0., 0., 1. );

    double p = 0., y = 0., r = 0.;
    if ( argc == 4 ) {

        p = atof( argv[1] );
        y = atof( argv[2] );
        r = atof( argv[3] );
    }

    std::cout << std::setprecision(3) << std::fixed;

    // specify pitch, yaw and roll of the target (degrees converted to radians)
```

```
    double pitch = va::rad( p );
    double yaw   = va::rad( y );
    double roll  = va::rad( r );

    // specify the rotation
    va::Rotation R( pitch, yaw, roll, va::XYZ );

    // apply the rotation to the basis vectors
    va::Vector ip = R * i;
    va::Vector jp = R * j;
    va::Vector kp = R * k;

    cout << "i = " << i << endl;
    cout << "j = " << j << endl;
    cout << "k = " << k << endl;

    cout << "ip = " << ip << endl;
    cout << "jp = " << jp << endl;
    cout << "kp = " << kp << endl;

    cout << endl << "Now we find the rotation" << endl;

    va::Rotation R1( i, j, k, ip, jp, kp );

    cout << "Applying the found rotation to i, j, k gives" << endl;

    ip = R1 * i;
    jp = R1 * j;
    kp = R1 * k;

    cout << "ip = " << ip << endl;
    cout << "jp = " << jp << endl;
    cout << "kp = " << kp << endl;

    cout << endl << "Now suppose we want to factor this rotation into a FATEPEN sequence" << endl;

    va::sequence s = va::factor( R1, va::XYZ );

    cout << "This gives:" << endl;
    cout << va::deg( s.first ) << endl;
    cout << va::deg( s.second ) << endl;
    cout << va::deg( s.third ) << endl;

    cout << endl << "Now for something completely different: Start with the non-unit vectors" << endl;

    va::Vector a( 1., 2., 3. ), b( -1., 2., 4. ), c( 4., 3., 9. );
    va::Vector ap, bp, cp;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;

    ap = R * a;
    bp = R * b;
    cp = R * c;

    cout << "ap = " << ap << endl;
    cout << "bp = " << bp << endl;
    cout << "cp = " << cp << endl;

    cout << endl << "Now we find the rotation that takes (a,b,c) to (ap,bp,cp)" << endl;

    va::Rotation R2( a, b, c, ap, bp, cp );

    cout << "Applying the found rotation to a, b, c gives" << endl;

    ap = R2 * a;
    bp = R2 * b;
    cp = R2 * c;

    cout << "ap = " << ap << endl;
    cout << "bp = " << bp << endl;
    cout << "cp = " << cp << endl;

    cout << endl << "Also, suppose we want to factor this rotation into a FATEPEN sequence" << endl;

    s = va::factor( R2, va::XYZ );

    cout << "This gives:" << endl;
    cout << va::deg( s.first ) << endl;
    cout << va::deg( s.second ) << endl;
    cout << va::deg( s.third ) << endl;

    return EXIT_SUCCESS;
}
```

We don't have to worry about whether the vectors are unit vectors or not, the Rotation class figures that out and performs the simpler method when it's able. Compiling and running this program with the command

```
./r2 60. -45. 15.
```

gives the following output:

```
i = 1.000 0.000 0.000
j = 0.000 1.000 0.000
k = 0.000 0.000 1.000
ip = 0.683 -0.462 0.566
jp = -0.183 0.641 0.745
kp = -0.707 -0.612 0.354

Now we find the rotation
Applying the found rotation to i, j, k gives
ip = 0.683 -0.462 0.566
jp = -0.183 0.641 0.745
kp = -0.707 -0.612 0.354

Now suppose we want to factor this rotation into a FATEPEN sequence
This gives:
60.000
-45.000
15.000

Now for something completely different: Start with the non-unit vectors
a = 1.000 2.000 3.000
b = -1.000 2.000 4.000
c = 4.000 3.000 9.000
ap = -1.804 -1.016 3.116
bp = -3.877 -0.704 2.339
cp = -4.181 -5.435 7.680

Now we find the rotation that takes (a,b,c) to (ap,bp,cp)
Applying the found rotation to a, b, c gives
ap = -1.804 -1.016 3.116
bp = -3.877 -0.704 2.339
cp = -4.181 -5.435 7.680

Also, suppose we want to factor this rotation into a FATEPEN sequence
This gives:
60.000
-45.000
15.000
```

4.3 Deflected Spall Cone from an Oblique Shot

When an overmatching penetrator perforates target armor, it generates a cone of spall fragments from the back of the armor. To better understand the threat this poses, individual spall fragments are registered as holes in a series of witness plates that are located some distance behind the target, typically 24 inches. In the case of an *oblique shot*, the target is at an obliquity angle α with respect to the shotline, and the witness plates are typically at half that angle, or $\alpha/2$, as depicted in Fig. 3. Although there are usually 5 plates in the witness pack during test conditions, here we will only consider the first plate of the pack.

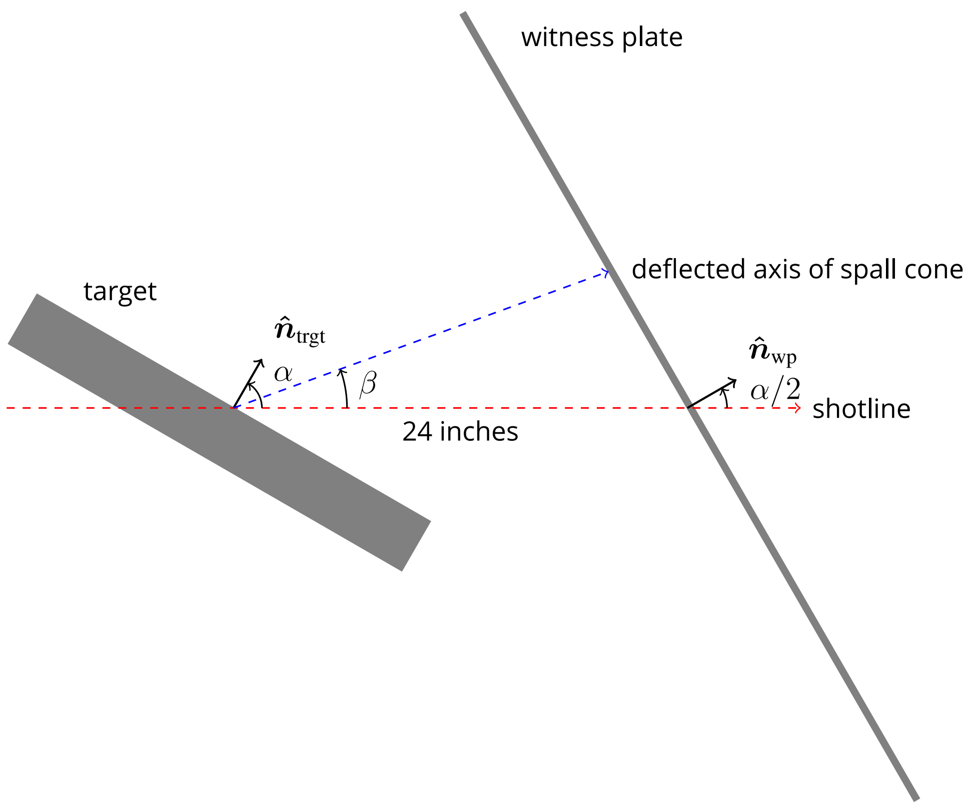

Fig. 3. Geometry for oblique shots

What we are going to do here is first make use of vector algebra to describe the coordinate system and the geometry of the test layout, along with the operational procedure to orient the target and witness plate. Following that, we will then show that it is easy to implement this into a C++ program by making use of the Vector and Rotation classes and by a straightforward translation of the vector equations.

Of course we need a coordinate system, but the coordinate system is constructed from the test geometry, not vice versa. Once the coordinate system is established, we can pretty much ignore the actual coordinates and just deal with vector quantities.

So we begin with the direction of the shot line and take this to be along the negative z-axis. Then we place the target plate in standard orientation, with its center at the origin, perpendicular to the shotline, its length along the x-axis, and its width along the y-axis. The same applies to the witness plate. These are just initial orientations; we will subsequently describe the procedure to put them in their final orientations and positions. Let $\hat{\boldsymbol{u}}_{\text{sl}}$ be a unit vector along the *initial* shotline. In our case, $\hat{\boldsymbol{u}}_{\text{sl}} = -\hat{\mathbf{k}}$, and remains fixed.

Next we orient the target with respect to the shotline. There are at least a couple of convenient ways to do this. We could specify directly the final orientation of the target, in which case the Rotation operator would be generated from the cross product from initial to final orientation.* Another method is to specify the rotation sequence that will take it from its initial orientation to its final orientation. We shall use that method here and adopt the FATEPEN[2,3] convention of first pitch about the x-axis, followed by yaw about the y-axis, and end with roll about the z-axis, so that

$$R_{\text{trgt}} = R_{\hat{\mathbf{i}}}(\phi_p) R_{\hat{\mathbf{j}}}(\phi_y) R_{\hat{\mathbf{k}}}(\phi_r), \tag{15}$$

where the order goes from right to left and is reversed since we rotate about fixed axes, not body axes.* Since the initial orientation is $-\hat{\mathbf{k}}$, the final orientation is

$$\hat{\boldsymbol{n}}_{\text{trgt}} = R_{\text{trgt}}(-\hat{\mathbf{k}}). \tag{16}$$

The target obliquity is labeled α, and

$$\alpha \equiv \theta_{\text{trgt}} = \cos^{-1}(|\hat{\boldsymbol{u}}_{\text{sl}} \cdot \hat{\boldsymbol{n}}_{\text{trgt}}|). \tag{17}$$

Next we want to orient the witness plate to be at half the obliquity angle of the target.

---

*Given an initial and a final orientation, $\hat{\boldsymbol{n}}_1$ and $\hat{\boldsymbol{n}}_2$, respectively, the Rotation operator that performs this rotation is $R_{\hat{\boldsymbol{n}}}(\theta)$, where

$$\hat{\boldsymbol{n}} = \frac{\hat{\boldsymbol{n}}_1 \times \hat{\boldsymbol{n}}_2}{\|\hat{\boldsymbol{n}}_1 \times \hat{\boldsymbol{n}}_2\|} \quad \text{and} \quad \theta = \cos^{-1}(|\hat{\boldsymbol{n}}_1 \cdot \hat{\boldsymbol{n}}_2|).$$

*See Appendix D for a justification.

This can be achieved with the aid of the spherical linear interpolation function[†]

$$\hat{\boldsymbol{n}}_{\text{wp}} = \text{slerp}(-\hat{\mathbf{k}}, \hat{\boldsymbol{n}}_{\text{trgt}}, 1/2), \tag{18}$$

which gives us a unit vector that is half the angle between $-\hat{\mathbf{k}}$ and $\hat{\boldsymbol{n}}_{\text{trgt}}$, and the witness plate rotation from standard orientation is

$$R_{\text{wp}} = R_{\hat{\boldsymbol{n}}}(\alpha/2) \quad \text{where} \quad \hat{\boldsymbol{n}} = \frac{(-\hat{\mathbf{k}}) \times \hat{\boldsymbol{n}}_{\text{wp}}}{\|(-\hat{\mathbf{k}}) \times \hat{\boldsymbol{n}}_{\text{wp}}\|}\,. \tag{19}$$

We might want to factor this rotation into a pitch-yaw-roll rotation sequence, since that would give us an operational procedure for achieving the orientation of the witness plate such that it is precisely at one-half the target orientation.

Now let us consider the deflection. If $\hat{\boldsymbol{u}}_{\text{sl}}$ and $\hat{\boldsymbol{n}}_{\text{trgt}}$ are not in the same direction (i.e., the target is at non-zero obliquity) then their vector cross product is non-zero and we can define the unit vector

$$\hat{\boldsymbol{w}} = \frac{\hat{\boldsymbol{u}}_{\text{sl}} \times \hat{\boldsymbol{n}}_{\text{trgt}}}{\|\hat{\boldsymbol{u}}_{\text{sl}} \times \hat{\boldsymbol{n}}_{\text{sl}}\|}. \tag{20}$$

The deflected shot line is obtained by rotating $\hat{\boldsymbol{u}}_{\text{sl}}$ about the unit vector $\hat{\boldsymbol{w}}$ through the rotation angle β, where $0 \le \beta \le \alpha$. Our notation for this rotation is $R_{\hat{\boldsymbol{w}}}(\beta)$. The witness plate has its center at the location $\boldsymbol{r}_c$ with respect to the rear of the target plate, typically 24 inches.

Now consider spall fragments that are coming off from the rear of the target. We can parametrize these fragments with the polar angle θ, measured from the $-z$-axis, and the azimuthal angle ϕ, measured from the x-axis.

$$\begin{aligned}\hat{\boldsymbol{u}}_{\text{frag}} &= \sin(\pi-\theta)\cos\phi\,\hat{\mathbf{i}} + \sin(\pi-\theta)\sin\phi\,\hat{\mathbf{j}} + \cos(\pi-\theta)\,\hat{\mathbf{k}} \\ &= \sin\theta\cos\phi\,\hat{\mathbf{i}} + \sin\theta\sin\phi\,\hat{\mathbf{j}} - \cos\theta\,\hat{\mathbf{k}} \end{aligned} \tag{21}$$

The spall fragments then get deflected according to the equation

$$\hat{\boldsymbol{u}}'_{\text{frag}} = R_{\hat{\boldsymbol{w}}}(\beta)\hat{\boldsymbol{u}}_{\text{frag}}. \tag{22}$$

The (deflected) fragment will travel in a straight line until it encounters the witness

[†]See Appendix G.

plate, so the equation for its trajectory is

$$\boldsymbol{r} = t\hat{\boldsymbol{u}}'_{\text{frag}}, \tag{23}$$

where the scalar $t > 0$ measures distance from the spall origin. The equation of the plane of the witness plate is the set of all points $\boldsymbol{r}$ that satisfy

$$(\boldsymbol{r} - \boldsymbol{r}_c) \cdot \hat{\boldsymbol{n}}_{\text{wp}} = 0. \tag{24}$$

Substituting Eq. 23 into Eq. 24 and solving for t gives for the distance

$$t = \frac{\boldsymbol{r}_c \cdot \hat{\boldsymbol{n}}_{\text{wp}}}{\hat{\boldsymbol{u}}'_{\text{frag}} \cdot \hat{\boldsymbol{n}}_{\text{wp}}}. \tag{25}$$

Hence, the hit point on the witness plate is

$$\boldsymbol{r}_{\text{hit}} = \left(\frac{\boldsymbol{r}_c \cdot \hat{\boldsymbol{n}}_{\text{wp}}}{\hat{\boldsymbol{u}}'_{\text{frag}} \cdot \hat{\boldsymbol{n}}_{\text{wp}}} \right) \hat{\boldsymbol{u}}'_{\text{frag}} \,. \tag{26}$$

To get the location of the hit point with respect to the witness plate center, we apply

$$R_{\text{wp}}^{-1}(\boldsymbol{r}_{\text{hit}} - \boldsymbol{r}_c), \tag{27}$$

where R_{wp}^{-1} is the inverse of the witness plate rotation, given by Eq. 19. The obliquity angle of each frag as it strikes the witness plate is

$$\theta_{\text{obl}} = \cos^{-1}(|\hat{\boldsymbol{u}}'_{\text{frag}} \cdot \hat{\boldsymbol{n}}_{\text{wp}}|). \tag{28}$$

The program in Listing 5 is an implementation of these equations and procedures.

Listing 5. oblique.cpp

```cpp
// oblique.cpp: geometry for oblique shots using a right-handed cartesian coordinate system
// shotline is fixed along the -z axis
// initial orientation of target is in the x-y plane with center at the origin
// initial orientation of witness plate is also in the x-y plane, but with center at z = -24 inches

#include "Rotation.h"
#include <iostream>
#include <cstdlib>
using namespace va;
using namespace std;

int main( int argc, char* argv[] ) {

   const int    N               = 300;   // number of frags on the spall cone
   const double WP_TRGT_OBL     = 0.5;   // ratio of witness plate obliquity to target obliquity (typically 0.5)
   const double CONE_HALF_ANGLE = 30.;   // cone half angle (deg)

   Vector ihat( 1., 0., 0. ), jhat( 0., 1., 0. ), khat( 0., 0., 1. );   // basis vectors
```

```
   Vector u_sl   = -khat;   // shotline fixed along -z axis
   Vector n_trgt = -khat;   // initial orientation of target
   Vector n_wp   = -khat;   // initial orientation of witness plate

   double pitch_t = 0., yaw_t = 0., roll_t = 0., def = 0.5;
   if ( argc == 5 ) {
      pitch_t = rad( atof( argv[1] ) );   // pitch (converted to rad) about x-axis
      yaw_t   = rad( atof( argv[2] ) );   // yaw (converted to rad) about y-axis
      roll_t  = rad( atof( argv[3] ) );   // roll (converted to rad) about z-axis
      def     = atof( argv[4] );          // deflection (dimensionless) ranges from 0 to 1
   }

   // orient the target by specifying pitch-yaw-roll rotation sequence (totally arbitrary)
   Rotation R_trgt( pitch_t, yaw_t, roll_t, XYZ );   // construct the rotation
   n_trgt = R_trgt * n_trgt;                         // apply the rotation to the target

   double alpha = angle( u_sl, n_trgt );    // compute obliquity of target (rad)

   // orient the witness plate using spherical linear interpolation
   Vector n = slerp( -khat, n_trgt, WP_TRGT_OBL );   // final orientation
   Rotation R_wp( ihat, 0. );                        // initialize rotation to zero
   if ( alpha > 0. && WP_TRGT_OBL > 0. )             // case of non-zero obliquity
      R_wp = Rotation( n_wp, n );                    // generate the rotation from the cross product
   n_wp = n;                                         // orient the witness plate
   Rotation R_wp_inv = -R_wp;                        // construct inverse rotation of witness plate

   // factor witness plate rotation into a pitch-yaw-roll rotation sequence and output
   sequence s = factor( R_wp, XYZ );
   clog << "Witness Plate Orientation (pitch-yaw-roll sequence):" << endl;
   clog << "pitch (deg) = " << deg( s.first )  << endl;
   clog << "yaw (deg)   = " << deg( s.second ) << endl;
   clog << "roll (deg)  = " << deg( s.third )  << endl;

   // output target and witness plate obliquity
   clog << "target obliquity (deg)      = " << deg( alpha ) << endl;
   clog << "wp obliquity (deg)          = " << deg( angle( u_sl, n_wp ) ) << endl;
   double beta  = def * alpha;   // deflection angle (rad)
   clog << "spall cone deflection (deg) = " << deg( beta ) << endl;

   Rotation R_def( ihat, 0. );       // rotation for deflection is initially zero
   Vector w = u_sl ^ n_trgt;         // w is cross product of shotline and target normal
   if ( w.mag() > 0. ) {             // w is nonzero iff target at obliquity
      w = normalize( w );            // make w into a unit vector
      R_def = Rotation( w, beta );   // rotation for deflection
   }

   Vector rc = -24. * khat;   // location of center of witness plate
   Vector u_frag, r_hit, r;
   double th = M_PI - rad( CONE_HALF_ANGLE ), ph, t;

   for ( int i = 0; i < N; i++ ) {   // generate frags on the spall cone

      ph     = 2. * M_PI * i / double( N );
      u_frag = sin( th ) * cos( ph ) * ihat + sin( th ) * sin( ph ) * jhat + cos( th ) * khat;
      u_frag = R_def * u_frag;
      t      = ( rc * n_wp ) / ( u_frag * n_wp );
      r_hit  = t * u_frag;

      r = R_wp_inv * ( r_hit - rc );
      cout << r.x() << "\t" << r.y() << endl;
   }

   return EXIT_SUCCESS;
}
```

Compiling and running this program with the command

```
./oblique 45. -34.2644 15. 0.5 > output
```

prints the following back to the screen:

```
Witness Plate Orientation (pitch-yaw-roll sequence):
pitch (deg) = 20.246
yaw (deg)   = -18.4381
roll (deg)  = 3.31976
target obliquity (deg)      = 54.2403
wp obliquity (deg)          = 27.1201
spall cone deflection (deg) = 27.1201
```

The `output` file contains the impact points on the witness plate from the `for` loop (lines 69–79 of oblique.cpp). Fig. 4 shows the resulting plots.

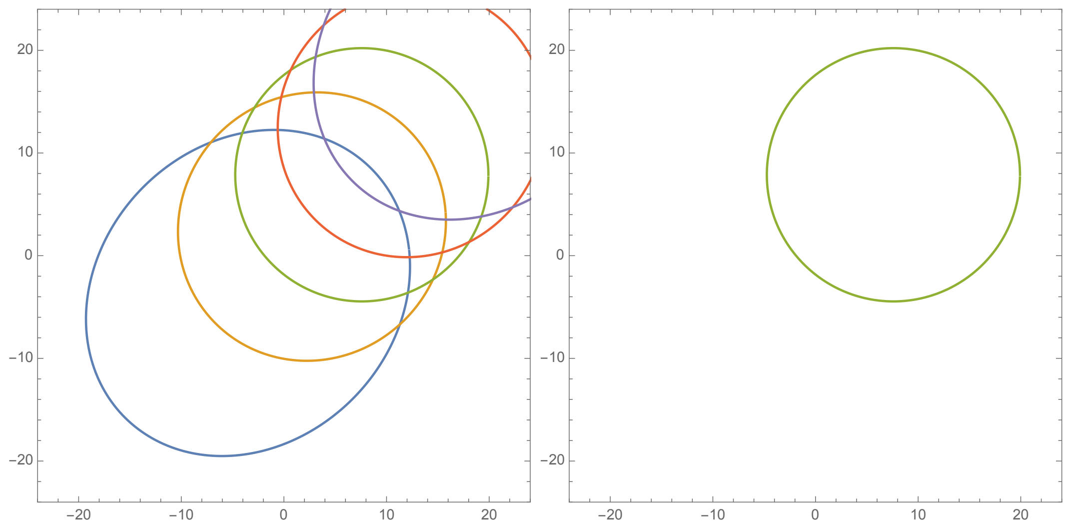

Fig. 4. Plots of the 30° (half-angle) spall cone on the witness plate for a range of deflections. On the left, the deflection parameter is 0, 0.25, 0.5, 0.75, and 1 as the ellipses go from lower left to upper right. On the right is the plot for a deflection of 0.5, which matches the witness plate obliquity and thus results in a circular spall cone. Target pitch, yaw, and roll are fixed at 45°, -34.2644°, and 15°, respectively giving a target obliquity for all theses cases of 54.2403°.

5. Conclusion

It should be clear from the examples that the Vector and Rotation classes provide robust support for performing 3D vector algebra in C++ programs. To make use of the Vector class, simply include the `Vector.h` class file and to make use of both the Vector and Rotation classes, simply include the `Rotation.h` class file in the C++ program.

6. References

1. Kuipers JB. Quaternions and rotation sequences: a primer with applications to orbits, aerospace, and virtual reality. Princeton (NJ): Princeton University Press; 2002.

2. Yatteau JD, Zernow RH, Recht GW, Edquist KT. FATEPEN. Ver. 3.0.0b. Terminal ballistic penetration model. Applied Research Associates (ARA) Project 4714, prepared for Naval Surface Warfare Center, Dahlgren, VA; 1999 Jan.

3. Yatteau JD, Zernow RH, Recht GW, Edquist KT. FATEPEN fast air target encounter penetration (Ver. 3.0.0) terminal ballistic penetration model. Littleton (CO): Applied Research Associates, Inc.; 2001 Sep. Rev. 2005 Feb 2. (Analyst's manual; vol. 1).

Appendix A. Vector Class

Listing A-1. Vector.h

```
// Vector.h: Definition & implementation of class for the algebra of 3D vectors
//           R. Saucier, February 2000 (Revised June 2016)

#ifndef VECTOR_3D_H
#define VECTOR_3D_H

#include <cstdlib>
#include <cassert>
#include <cmath>
#include <iostream>

namespace va {   // vector algeba namespace

enum rep { CART, POLAR };          // vector representation (cartesian or polar)
enum comp { X, Y, Z };             // for referencing cartesian components
const double R2D( 180. / M_PI );   // for converting radians to degrees
const double D2R( M_PI / 180. );   // for converting degrees to radians
inline double deg( const double rad ) { return rad * R2D; }   // convert radians to degrees
inline double rad( const double deg ) { return deg * D2R; }   // convert degrees to radians
// IEEE 754 standard has 53 significant bits or 15 decimal digits of accuracy, so anything smaller is not significant
static const long double TOL = 1.e-15;

class Vector {

// friends list
// overloaded arithmetic operators

   friend Vector operator+( const Vector& a, const Vector& b ) {   // addition

      return Vector( a._x + b._x, a._y + b._y, a._z + b._z );
   }

   friend Vector operator-( const Vector& a, const Vector& b ) {   // subtraction

     return Vector( a._x - b._x, a._y - b._y, a._z - b._z );
   }

   friend Vector operator*( const Vector& v, double s ) {   // right multiply by scalar

      return Vector( s * v._x, s * v._y, s * v._z );
   }

   friend Vector operator*( double s, const Vector& v ) {   // left multiply by scalar

      return Vector( s * v._x, s * v._y, s * v._z );
   }

   friend Vector operator/( const Vector& v, double s ) {   // division by scalar

      assert( s != 0. );
      return Vector( v._x / s, v._y / s, v._z / s );
   }

// overloaded algebraic operators

   friend inline double operator*( const Vector& a, const Vector& b ) { // dot product

      double c( a._x * b._x +
                a._y * b._y +
                a._z * b._z );
      if ( fabs(c) < TOL ) c = 0.0L;   // set precision to be no more than 15 digits
      return c;
   }

   friend inline Vector operator^( const Vector& a, const Vector& b ) { // cross product

      return Vector( a._y * b._z - a._z * b._y,
                     a._z * b._x - a._x * b._z,
                     a._x * b._y - a._y * b._x );
   }

// access functions

   friend double x( const Vector& v ) {   // x-coordinate

      return v._x;
   }

   friend double y( const Vector& v ) {   // y-coordinate

      return v._y;
   }

```

```
   friend double z( const Vector& v ) {   // z-coordinate

      return v._z;
   }

   friend double r( const Vector& v ) {   // magnitude of vector

      return v._mag();
   }

   friend double theta( const Vector& v ) {   // polar angle (radians)

      return v.theta();
   }

   friend double phi( const Vector& v ) {   // azimuthal angle (radians)

      return v.phi();
   }

   friend double norm( const Vector& v ) {   // norm or magnitude

      return v._mag();
   }

   friend double mag( const Vector& v ) {   // magnitude

      return v._mag();
   }

   friend double scalar( const Vector& v ) {   // magnitude

      return v._mag();
   }

   friend double angle( const Vector& a, const Vector& b ) { // angle (radians) between vectors

      double s = a.unit() * b.unit();
      if ( s >= 1. )
         return 0.;
      else if ( s <= -1. )
         return M_PI;
      else
         return acos( s );
   }

   friend Vector unit( const Vector& v ) {   // unit vector in same direction

      return v.unit();
   }

   friend Vector normalize( const Vector& v ) {   // returns a unit vector in same direction

      return v.unit();
   }

   friend double dircos( const Vector& v, const comp& i ) {   // direction cosine

      return v.dircos( i );
   }

   friend Vector proj( const Vector& a,      // projection of first vector
                       const Vector& b ) {   // along second vector

      return a.proj( b );
   }

// overloaded stream operators

   friend std::istream& operator>>( std::istream& is, Vector& v ) {   // input vector

      return is >> v._x >> v._y >> v._z;
   }

   friend std::ostream& operator<<( std::ostream& os, const Vector& v ) {   // output vector

      Vector a( v );
      a._set_precision();
      return os << a._x << " " << a._y << " " << a._z;
   }

public:

   Vector( double x, double y, double z,   // constructor (cartesian or polar
           rep mode = CART ) {             // with cartesian as default)
```

```
      if ( mode == CART ) {   // cartesian form
         this->_x = x;
         this->_y = y;
         this->_z = z;
      }
      else if ( mode == POLAR )   // polar form
         _setCartesian( x, y, z );
      else {
         std::cerr << "Vector: mode must be either CART or POLAR" << std::endl;
         exit( EXIT_FAILURE );
      }
   }

   Vector( void ) : _x( 0. ), _y( 0. ), _z( 0. ) {   // default constructor
   }

  ~Vector( void ) {   // default destructor
   }

   Vector( const Vector& v ) : _x( v._x ),    // copy constructor
                               _y( v._y ),
                               _z( v._z ) {
   }

   Vector& operator=( const Vector& v ) {   // assignment operator

      if ( this != &v ) {
         _x = v._x;
         _y = v._y;
         _z = v._z;
      }
      return *this;
   }

// overloaded arithmetic operators

   Vector& operator+=( const Vector& v ) {   // addition assignment

      _x += v._x;
      _y += v._y;
      _z += v._z;
      return *this;
   }

   Vector& operator-=( const Vector& v ) {   // subtraction assignment

      _x -= v._x;
      _y -= v._y;
      _z -= v._z;
      return *this;
   }

   Vector& operator*=( double s ) {   // multiplication assignment

      _x *= s;
      _y *= s;
      _z *= s;
      return *this;
   }

   Vector& operator/=( double s ) {   // division assignment

      assert( s != 0. );
      _x /= s;
      _y /= s;
      _z /= s;
      return *this;
   }

   Vector operator-( void ) {   // negative of a vector

      return Vector( -_x, -_y, -_z );
   }

   const double& operator[]( comp i ) const {   // index operator (component)

      if ( i == X ) return _x;
      if ( i == Y ) return _y;
      if ( i == Z ) return _z;
      std::cerr << "Vector: Array index out of range; must be X, Y, or Z" << std::endl;
      exit( EXIT_FAILURE );
   }

// access functions
```

```
   double x( void ) const {   // x-component

      return _x;
   }

   double y( void ) const {   // y-component

      return _y;
   }

   double z( void ) const {   // z-component

      return _z;
   }

   double r( void ) const {   // magnitude

      return _mag();
   }

   double theta( void ) const {   // polar angle (radians)

      return _theta();
   }

   double phi( void ) const {   // azimuthal angle (radians)

      return _phi();
   }

   double norm( void ) const {   // norm or magnitude

      return _mag();
   }

   double mag( void ) const {   // magnitude

      return _mag();
   }

   operator double( void ) const {   // conversion operator to return magnitude

      return _mag();
   }

// utility functions

   Vector unit( void ) const {   // returns a unit vector

      double m = _mag();
      if ( m > 0. ) return Vector( _x / m, _y / m, _z / m );
      std::cerr << "Vector: Cannot make a unit vector from a null vector" << std::endl;
      exit( EXIT_FAILURE );
   }

   Vector normalize( void ) const {   // synonym for unit

      return unit();
   }

   double dircos( const comp& i ) const {   // direction cosine

      if ( i == X ) return _x / _mag();
      if ( i == Y ) return _y / _mag();
      if ( i == Z ) return _z / _mag();
      std::cerr << "Vector dircos: comp out of range; must be X, Y, or Z" << std::endl;
      exit( EXIT_FAILURE );
   }

   Vector proj( const Vector& e ) const {   // projects onto the given vector

      Vector u = e.unit();
      return u * ( *this * u );
   }

   Vector& rotate( const Vector& a, double angle ) {   // rotates vector about given axial vector, a, through the given
         angle

      Vector a_hat = a.unit();        // unit vector along the axial vector
      Vector cross = a_hat ^ *this;   // cross product of a_hat with the given vector
      return *this += sin( angle ) * cross + ( 1. - cos( angle ) ) * ( a_hat ^ cross );
   }

   Vector& rot( const Vector& a, double angle ) {   // synonym for rotate
```

```

      return rotate( a, angle );
   }

private:

   double _x, _y, _z;   // cartesian representation

   double _mag( void ) const {   // compute the magnitude

      double mag = sqrt( _x * _x + _y * _y + _z * _z );
      if ( mag < TOL ) mag = 0.0L;
      return mag;
   }

   double _theta( void ) const {   // compute the polar angle (radians)

      if ( _z != 0. ) return atan2( sqrt( _x * _x + _y * _y ), _z );
      else            return M_PI_2;
   }

   double _phi( void ) const {   // compute the azimuthal angle (radians)

      if ( _x != 0. )
         return atan2( _y, _x );
      else {
         if ( _y > 0   )
            return M_PI_2;
         else if ( _y == 0. )
            return 0.;
         else
            return -M_PI_2;
      }
   }

   // set cartesian representation from polar
   void _setCartesian( double r, double theta, double phi ) {

      _x = r * sin( theta ) * cos( phi );
      _y = r * sin( theta ) * sin( phi );
      _z = r * cos( theta );
   }

   Vector _set_precision( void ) {   // no more than 15 digits of precision

      if ( fabs(_x) < TOL ) _x = 0.0L;
      if ( fabs(_y) < TOL ) _y = 0.0L;
      if ( fabs(_z) < TOL ) _z = 0.0L;

      return *this;
   }
};
   // declaration of friends
   Vector operator+( const Vector& a, const Vector& b );
   Vector operator-( const Vector& a, const Vector& b );
   Vector operator*( const Vector& v, double s );
   Vector operator*( double s, const Vector& v );
   Vector operator/( const Vector& v, double s );
   double operator*( const Vector& a, const Vector& b );
   Vector operator^( const Vector& a, const Vector& b );
   double x( const Vector& v );
   double y( const Vector& v );
   double z( const Vector& v );
   double r( const Vector& v );
   double theta( const Vector& v );
   double phi( const Vector& v );
   double mag( const Vector& v );
   double scalar( const Vector& v );
   double angle( const Vector& a, const Vector& b );
   Vector unit( const Vector& v );
   Vector normalize( const Vector& v );
   double dircos( const Vector& v, const comp& i );
   Vector proj( const Vector& a, const Vector& b );
   std::istream& operator>>( std::istream& is, Vector& v );
   std::ostream& operator<<( std::ostream& os, const Vector& v );
}  // vector algebra namespace
#endif
```

Notice that the class is enclosed in a va namespace, so that Vectors are declared by va::Vector. Table A-1 provides a reference sheet for basic usage.

Table A-1. **Vector: A C++ class for 3-dimensional vector algebra—reference sheet**

Operation	Mathematical notation	Vector class
Definition	Let $\boldsymbol{v}$ be an unspecified vector.	`Vector v;`
	Let $\boldsymbol{a}$ be the cartesian vector (1,2,3).	`Vector a(1,2,3);` *or* `Vector a(1,2,3,CART);`
	Let $\boldsymbol{b}$ be the polar vector (r,θ,ϕ).[a]	`Vector b(r,th,ph,POLAR);`
Input vector $\boldsymbol{a}$	*n/a*	`cin >> a;`
Output vector $\boldsymbol{a}$	*n/a*	`cout << a;`
Cartesian representation	Let $\boldsymbol{a} = x\hat{\mathbf{\imath}} + y\hat{\mathbf{\jmath}} + z\hat{\mathbf{k}}$.[b]	`Vector a(x,y,z);` *or* `Vector a(x,y,z,CART);`
Polar representation	Let $\boldsymbol{a} = (r, \theta, \phi)$.[a]	`Vector a(r,th,ph,POLAR);`
Assign one vector to another	Let $\boldsymbol{b} = \boldsymbol{a}$ *or* $\boldsymbol{b} \Leftarrow \boldsymbol{a}$	`b = a;` *or* `b(a);`
Components of vector $\boldsymbol{a}$	a_x, a_y, a_z	`a.x(), a.y(), a.z()` *or* `x(a), y(a), z(a)` *or* `a[X], a[Y], a[Z]`
	r, θ, ϕ	`a.r(), a.theta(), a.phi()` *or* `r(a), theta(a), phi(a)`
Direction cosines	$\boldsymbol{v} \cdot \hat{\mathbf{\imath}}/\lVert\boldsymbol{v}\rVert$, $\boldsymbol{v} \cdot \hat{\mathbf{\jmath}}/\lVert\boldsymbol{v}\rVert$, $\boldsymbol{v} \cdot \hat{\mathbf{k}}/\lVert\boldsymbol{v}\rVert$	`v.dircos(X); ...` *or* `dircos(v,X); ...`
Vector addition	$\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$	`c = a + b;`
Addition assignment	$\boldsymbol{b} \Leftarrow \boldsymbol{b} + \boldsymbol{a}$	`b += a;`
Vector subtraction	$\boldsymbol{c} = \boldsymbol{a} - \boldsymbol{b}$	`c = a - b;`
Subtraction assignment	$\boldsymbol{b} \Leftarrow \boldsymbol{b} - \boldsymbol{a}$	`b -= a;`
Multiplication by a scalar s	$\boldsymbol{b} = s\,\boldsymbol{a}$ *or* $\boldsymbol{b} = \boldsymbol{a}\,s$	`b = s * a;` *or* `b = a * s;`
Multiplication assignment	$\boldsymbol{a} \Leftarrow s\,\boldsymbol{a}$ *or* $\boldsymbol{a} \Leftarrow \boldsymbol{a}\,s$	`a *= s;`
Dot (scalar) product	$\mathrm{c} = \boldsymbol{a} \cdot \boldsymbol{b}$	`c = a * b;`
Cross (vector) product	$\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$	`c = a ^ b;`
Negative of a vector	$-\boldsymbol{v}$	`-v;`
Norm, or magnitude, of a vector	$\lVert\boldsymbol{v}\rVert$	`v.norm();` *or* `norm(v);` *or* `v.mag();` *or* `mag(v);` *or* `v.r();` *or* `v.scalar();`
Angle between two vectors	$\theta = \cos^{-1}\left(\dfrac{\boldsymbol{a} \cdot \boldsymbol{b}}{\lVert\boldsymbol{a}\rVert\,\lVert\boldsymbol{b}\rVert}\right)$	`angle(a, b);`
Normalize a vector	$\hat{\boldsymbol{u}} = \boldsymbol{v}/\lVert\boldsymbol{v}\rVert$	`u = v.normalize();`[c] *or* `u = normalize(v);` *or* `u = v.unit();` *or* `u = unit(v);`
Projection of $\boldsymbol{a}$ along $\boldsymbol{b}$	$\left(\boldsymbol{a} \cdot \dfrac{\boldsymbol{b}}{\lVert\boldsymbol{b}\rVert}\right)\dfrac{\boldsymbol{b}}{\lVert\boldsymbol{b}\rVert}$	`proj(a, b);` *or* `a.proj(b);`
Rotate vector $\boldsymbol{a}$ about the axial vector $\hat{\boldsymbol{u}}$ through the angle θ	$\boldsymbol{a} + \hat{\boldsymbol{u}} \times \boldsymbol{a} \sin\theta + \hat{\boldsymbol{u}} \times (\hat{\boldsymbol{u}} \times \boldsymbol{a})(1 - \cos\theta)$	`a.rotate(u, theta);` *or* `a.rot(u, theta);`

[a] r is the magnitude, θ is the polar angle measured from the z-axis, and ϕ is the azimuthal angle measured from the x-axis to the plane that contains the vector and the z-axis. The angle θ and ϕ are in radians. Use `rad(deg)` to convert degrees to radians and `deg(rad)` to convert radians to degrees.

[b] $\hat{\mathbf{\imath}}$, $\hat{\mathbf{\jmath}}$, and $\hat{\mathbf{k}}$ are unit vectors along the x-axis, y-axis, and z-axis, respectively.

[c] `normalize` does not change the vector it is invoked on; it merely returns the vector divided by its norm.

INTENTIONALLY LEFT BLANK.

Appendix B. Rotation Class

Listing B-1. Rotation.h

```
// Rotation.h: Rotation class definition for the algebra of 3D rotations
// Ref: Kuipers, J. B., Quaternions and Rotation Sequences, Princeton, 1999.
//      Altmann, S. L., Rotations, Quaternions, and Double Groups, 1986.
//      Doran & Lasenby, Geometric Algebra for Physicists, Cambridge, 2003.
// R. Saucier, March 2005 (Last revised June 2016)

#ifndef ROTATION_H
#define ROTATION_H

#include "Vector.h"
#include "Random.h"
#include <iostream>

namespace va {   // vector algebra namespace

const Vector DEFAULT_UNIT_VECTOR( 0., 0., 1. );   // arbitrarily choose k
const double DEFAULT_ROTATION_ANGLE( 0. );        // arbitrarily choose 0
const double TWO_PI( 2. * M_PI );

enum ORDER {   // order of rotation sequence about body axes

   // six distinct principal axes factorizations
   ZYX,   // first about z-axis, second about y-axis and third about x-axis
   XYZ,   // first about x-axis, second about y-axis and third about z-axis
   YXZ,   // first about y-axis, second about x-axis and third about z-axis
   ZXY,   // first about z-axis, second about x-axis and third about y-axis
   XZY,   // first about x-axis, second about z-axis and third about y-axis
   YZX,   // first about y-axis, second about z-axis and third about x-axis

   // six repeated principal axes factorizations
   ZYZ,   // first about z-axis, second about y-axis and third about z-axis
   ZXZ,   // first about z-axis, second about x-axis and third about z-axis
   YZY,   // first about y-axis, second about z-axis and third about y-axis
   YXY,   // first about y-axis, second about x-axis and third about y-axis
   XYX,   // first about x-axis, second about y-axis and third about x-axis
   XZX    // first about x-axis, second about z-axis and third about x-axis
};

struct quaternion {   // q = w + v, where w is scalar part and v is vector part

   quaternion( void ) {
   }

   quaternion( double scalar, Vector vector ) : w( scalar ), v( vector ) {
   }

  ~quaternion( void ) {
   }

   // overloaded multiplication of two quaternions
   friend quaternion operator*( const quaternion& q1, const quaternion& q2 ) {

      return quaternion(
         ( q1.w * q2.w ) - ( q1.v * q2.v ),                       // scalar part
         ( q1.w * q2.v ) + ( q2.w * q1.v ) + ( q1.v ^ q2.v )    // vector part
      );
   }

   double w;   // scalar part
   Vector v;   // vector part (actually a bivector disguised as a vector)
};

struct sequence {   // rotation sequence about three principal body axes

   sequence( void ) {
   }

   sequence( double phi_1, double phi_2, double phi_3 ) : first( phi_1 ), second( phi_2 ), third( phi_3 ) {
   }

  ~sequence( void ) {
   }

   // factor quaternion into Euler rotation sequence about three distinct principal axes
   sequence factor( const quaternion& p, ORDER order ) {

      double p0 = p.w;
      double p1 = x( p.v );
      double p2 = y( p.v );
      double p3 = z( p.v );

      double phi_1, phi_2, phi_3;
      if ( order == ZYX ) {   // distinct principal axes zyx
```

Approved for public release; distribution is unlimited.

```
        double A = p0 * p1 + p2 * p3;
        double B = ( p2 - p0 ) * ( p2 + p0 );
        double D = ( p1 - p3 ) * ( p1 + p3 );

        phi_3 = atan( - 2. * A / ( B + D ) );
        double c0 = cos( 0.5 * phi_3 );
        double c1 = sin( 0.5 * phi_3 );

        double q0 = p0 * c0 + p1 * c1;
        //double q1 = p1 * c0 - p0 * c1;
        double q2 = p2 * c0 - p3 * c1;
        double q3 = p3 * c0 + p2 * c1;

        phi_1 = 2. * atan( q3 / q0 );
        phi_2 = 2. * atan( q2 / q0 );
    }
    else if ( order == XYZ ) {   // distinct principal axes xyz

        double A = p1 * p2 - p0 * p3;
        double B = ( p1 - p3 ) * ( p1 + p3 );
        double D = ( p0 - p2 ) * ( p0 + p2 );

        phi_3 = atan( - 2. * A / ( B + D ) );
        double c0 = cos( 0.5 * phi_3 );
        double c3 = sin( 0.5 * phi_3 );

        double q0 = p0 * c0 + p3 * c3;
        double q1 = p1 * c0 - p2 * c3;
        double q2 = p2 * c0 + p1 * c3;
        //double q3 = p3 * c0 - p0 * c3;

        phi_1 = 2. * atan( q1 / q0 );
        phi_2 = 2. * atan( q2 / q0 );
    }
    else if ( order == YXZ ) {   // distinct principal axes yxz

        double A = p1 * p2 + p0 * p3;
        double B = p1 * p1 + p3 * p3;
        double D = -( p0 * p0 + p2 * p2 );

        phi_3 = atan( - 2. * A / ( B + D ) );
        double c0 = cos( 0.5 * phi_3 );
        double c3 = sin( 0.5 * phi_3 );

        double q0 = p0 * c0 + p3 * c3;
        double q1 = p1 * c0 - p2 * c3;
        double q2 = p2 * c0 + p1 * c3;
        //double q3 = p3 * c0 - p0 * c3;

        phi_1 = 2. * atan( q2 / q0 );
        phi_2 = 2. * atan( q1 / q0 );
    }
    else if ( order == ZXY ) {   // distinct principal axes zxy

        double A = p1 * p3 - p0 * p2;
        double B = ( p0 - p1 ) * ( p0 + p1 );
        double D = ( p3 - p2 ) * ( p3 + p2 );

        phi_3 = atan( - 2. * A / ( B + D ) );
        double c0 = cos( 0.5 * phi_3 );
        double c2 = sin( 0.5 * phi_3 );

        double q0 = p0 * c0 + p2 * c2;
        double q1 = p1 * c0 + p3 * c2;
        //double q2 = p2 * c0 - p0 * c2;
        double q3 = p3 * c0 - p1 * c2;

        phi_1 = 2. * atan( q3 / q0 );
        phi_2 = 2. * atan( q1 / q0 );
    }
    else if ( order == XZY ) {   // distinct principal axes xzy

        double A = p1 * p3 + p0 * p2;
        double B = -( p0 * p0 + p1 * p1 );
        double D = p2 * p2 + p3 * p3;

        phi_3 = atan( - 2. * A / ( B + D ) );
        double c0 = cos( 0.5 * phi_3 );
        double c2 = sin( 0.5 * phi_3 );

        double q0 = p0 * c0 + p2 * c2;
        double q1 = p1 * c0 + p3 * c2;
        //double q2 = p2 * c0 - p0 * c2;
        double q3 = p3 * c0 - p1 * c2;
```

```

            phi_1 = 2. * atan( q1 / q0 );
            phi_2 = 2. * atan( q3 / q0 );
        }
        else if ( order == YZX ) {   // distinct principal axes yzx

            double A = p2 * p3 - p0 * p1;
            double B = p0 * p0 + p2 * p2;
            double D = -( p1 * p1 + p3 * p3 );

            phi_3 = atan( - 2. * A / ( B + D ) );
            double c0 = cos( 0.5 * phi_3 );
            double c1 = sin( 0.5 * phi_3 );

            double q0 = p0 * c0 + p1 * c1;
            //double q1 = p1 * c0 - p0 * c1;
            double q2 = p2 * c0 - p3 * c1;
            double q3 = p3 * c0 + p2 * c1;

            phi_1 = 2. * atan( q2 / q0 );
            phi_2 = 2. * atan( q3 / q0 );
        }
        else if ( order == ZYZ ) {   // repeated principal axes zyz

            double A = p0 * p1 + p2 * p3;
            double B = -2. * p0 * p2;
            double D = 2. * p1 * p3;

            phi_3 = atan( -2. * A / ( B + D ) );
            double c0 = cos( 0.5 * phi_3 );
            double c3 = sin( 0.5 * phi_3 );

            double q0 = p0 * c0 + p3 * c3;
            //double q1 = p1 * c0 - p2 * c3;
            double q2 = p2 * c0 + p1 * c3;
            double q3 = p3 * c0 - p0 * c3;

            phi_1 = 2. * atan( q3 / q0 );
            phi_2 = 2. * atan( q2 / q0 );
        }
        else if ( order == ZXZ ) {   // repeated principal axes zxz

            double A = p0 * p2 - p1 * p3;
            double B = 2. * p0 * p1;
            double D = 2. * p2 * p3;

            phi_3 = atan( -2. * A / ( B + D ) );
            double c0 = cos( 0.5 * phi_3 );
            double c3 = sin( 0.5 * phi_3 );

            double q0 = p0 * c0 + p3 * c3;
            double q1 = p1 * c0 - p2 * c3;
            //double q2 = p2 * c0 + p1 * c3;
            double q3 = p3 * c0 - p0 * c3;

            phi_1 = 2. * atan( q3 / q0 );
            phi_2 = 2. * atan( q1 / q0 );
        }
        else if ( order == YZY ) {   // repeated principal axes yzy

            double A = p0 * p1 - p2 * p3;
            double B = p0 * p3 + p1 * p2;

            phi_3 = atan( -A / B );
            double c0 = cos( 0.5 * phi_3 );
            double c2 = sin( 0.5 * phi_3 );

            double q0 = p0 * c0 + p2 * c2;
            //double q1 = p1 * c0 + p3 * c2;
            double q2 = p2 * c0 - p0 * c2;
            double q3 = p3 * c0 - p1 * c2;

            phi_1 = 2. * atan( q2 / q0 );
            phi_2 = 2. * atan( q3 / q0 );
        }
        else if ( order == YXY ) {   // repeated principal axes yxy

            double A = p0 * p3 + p1 * p2;
            double B = -2. * p0 * p1;
            double D = 2. * p2 * p3;

            phi_3 = atan( -2. * A / ( B + D ) );
            double c0 = cos( 0.5 * phi_3 );
            double c2 = sin( 0.5 * phi_3 );

```

```
         double q0 = p0 * c0 + p2 * c2;
         double q1 = p1 * c0 + p3 * c2;
         double q2 = p2 * c0 - p0 * c2;
         //double q3 = p3 * c0 - p1 * c2;

         phi_1 = 2. * atan( q2 / q0 );
         phi_2 = 2. * atan( q1 / q0 );
      }
      else if ( order == XYX ) {   // repeated principal axes xyx

         double A = p0 * p3 - p1 * p2;
         double B = p0 * p2 + p1 * p3;

         phi_3 = atan( -A / B );
         double c0 = cos( 0.5 * phi_3 );
         double c1 = sin( 0.5 * phi_3 );

         double q0 = p0 * c0 + p1 * c1;
         double q1 = p1 * c0 - p0 * c1;
         double q2 = p2 * c0 - p3 * c1;
         //double q3 = p3 * c0 + p2 * c1;

         phi_1 = 2. * atan( q1 / q0 );
         phi_2 = 2. * atan( q2 / q0 );
      }
      else if ( order == XZX ) {   // repeated principal axes xzx

         double A = p0 * p2 + p1 * p3;
         double B = -p0 * p3 + p1 * p2;

         phi_3 = atan( -A / B );
         double c0 = cos( 0.5 * phi_3 );
         double c1 = sin( 0.5 * phi_3 );

         double q0 = p0 * c0 + p1 * c1;
         double q1 = p1 * c0 - p0 * c1;
         //double q2 = p2 * c0 - p3 * c1;
         double q3 = p3 * c0 + p2 * c1;

         phi_1 = 2. * atan( q1 / q0 );
         phi_2 = 2. * atan( q3 / q0 );
      }
      else {
         std::cerr << "ERROR in Rotation: invalid sequence order: " << order << std::endl;
         exit( EXIT_FAILURE );
      }
      return sequence( phi_1, phi_2, phi_3 );
   }

   double first,    // about first body axis
          second,   // about second body axis
          third;    // about third body axis
};   // end struct sequence

struct matrix {   // all matrices here are rotations in three-space

   // overloaded multiplication of two matrices
   // (defined as a convenience to the user; not used in Rotation class)
   friend matrix operator*( const matrix& A, const matrix& B ) {

      matrix C;
      C.a11 = A.a11 * B.a11 + A.a12 * B.a21 + A.a13 * B.a31;
      C.a12 = A.a11 * B.a12 + A.a12 * B.a22 + A.a13 * B.a32;
      C.a13 = A.a11 * B.a13 + A.a12 * B.a23 + A.a13 * B.a33;

      C.a21 = A.a21 * B.a11 + A.a22 * B.a21 + A.a23 * B.a31;
      C.a22 = A.a21 * B.a12 + A.a22 * B.a22 + A.a23 * B.a32;
      C.a23 = A.a21 * B.a13 + A.a22 * B.a23 + A.a23 * B.a33;

      C.a31 = A.a31 * B.a11 + A.a32 * B.a21 + A.a33 * B.a31;
      C.a32 = A.a31 * B.a12 + A.a32 * B.a22 + A.a33 * B.a32;
      C.a33 = A.a31 * B.a13 + A.a32 * B.a23 + A.a33 * B.a33;

      return C;
   }

   // transpose of a matrix
   // (defined as a convenience to the user; not used in Rotation class)
   friend matrix transpose( const matrix& A ) {

      matrix B;
      B.a11 = A.a11;
      B.a12 = A.a21;
      B.a13 = A.a31;

```

```
      B.a21 = A.a12;
      B.a22 = A.a22;
      B.a23 = A.a32;

      B.a31 = A.a13;
      B.a32 = A.a23;
      B.a33 = A.a33;

      return B;
   }

   // inverse of a matrix
   // (defined as a convenience to the user; not used in Rotation class)
   friend matrix inverse( const matrix& A ) {

      double det = A.a11 * ( A.a22 * A.a33 - A.a23 * A.a32 ) +
                   A.a12 * ( A.a23 * A.a31 - A.a21 * A.a33 ) +
                   A.a13 * ( A.a21 * A.a32 - A.a22 * A.a31 );
      assert( det != 0. );
      matrix B;
      B.a11 = +( A.a22 * A.a33 - A.a23 * A.a32 ) / det;
      B.a12 = -( A.a12 * A.a33 - A.a13 * A.a32 ) / det;
      B.a13 = +( A.a12 * A.a23 - A.a13 * A.a22 ) / det;

      B.a21 = -( A.a21 * A.a33 - A.a23 * A.a31 ) / det;
      B.a22 = +( A.a11 * A.a33 - A.a13 * A.a31 ) / det;
      B.a23 = -( A.a11 * A.a23 - A.a13 * A.a21 ) / det;

      B.a31 = +( A.a21 * A.a32 - A.a22 * A.a31 ) / det;
      B.a32 = -( A.a11 * A.a32 - A.a12 * A.a31 ) / det;
      B.a33 = +( A.a11 * A.a22 - A.a12 * A.a21 ) / det;

      return B;
   }

   // returns the matrix with no more than 15 decimal digit accuracy
   friend matrix set_precision( const matrix& A ) {

      matrix B( A );
      if ( fabs(B.a11) < TOL ) B.a11 = 0.0L;
      if ( fabs(B.a12) < TOL ) B.a12 = 0.0L;
      if ( fabs(B.a13) < TOL ) B.a13 = 0.0L;

      if ( fabs(B.a21) < TOL ) B.a21 = 0.0L;
      if ( fabs(B.a22) < TOL ) B.a22 = 0.0L;
      if ( fabs(B.a23) < TOL ) B.a23 = 0.0L;

      if ( fabs(B.a31) < TOL ) B.a31 = 0.0L;
      if ( fabs(B.a32) < TOL ) B.a32 = 0.0L;
      if ( fabs(B.a33) < TOL ) B.a33 = 0.0L;

      return B;
   }

   // convenient matrix properties, but not essential to the Rotation class

   friend double tr( const matrix& A ) {   // trace of a matrix

      return A.a11 + A.a22 + A.a33;
   }

   friend double det( const matrix& A ) {   // determinant of a matrix

      return A.a11 * ( A.a22 * A.a33 - A.a23 * A.a32 ) +
             A.a12 * ( A.a23 * A.a31 - A.a21 * A.a33 ) +
             A.a13 * ( A.a21 * A.a32 - A.a22 * A.a31 );
   }

   friend Vector eigenvector( const matrix& A ) {   // eigenvector of a matrix

      return unit( ( A.a32 - A.a23 ) * Vector( 1., 0., 0. ) +
                   ( A.a13 - A.a31 ) * Vector( 0., 1., 0. ) +
                   ( A.a21 - A.a12 ) * Vector( 0., 0., 1. ) );
   }

   friend double angle( const matrix& A ) {   // angle of rotation

      return acos( 0.5 * ( A.a11 + A.a22 + A.a33 - 1. ) );
   }

   Vector eigenvector( void ) {   // axis of rotation

      return unit( ( a32 - a23 ) * Vector( 1., 0., 0. ) +
                   ( a13 - a31 ) * Vector( 0., 1., 0. ) +
                   ( a21 - a12 ) * Vector( 0., 0., 1. ) );
```

```
   }

   double tr( void ) {   // trace of the matrix

      return a11 + a22 + a33;
   }

   double angle( void ) {   // angle of rotation

      return acos( 0.5 * ( a11 + a22 + a33 - 1. ) );
   }

   double a11, a12, a13,   // 1st row
          a21, a22, a23,   // 2nd row
          a31, a32, a33;   // 3rd row
};   // end struct matrix

class Rotation {

   // friends list

   // overloaded multiplication of two successive rotations (using quaternions)
   // notice the order is important; first the right is applied and then the left
   friend Rotation operator*( const Rotation& R1, const Rotation& R2 ) {

      return Rotation( to_quaternion( R1 ) * to_quaternion( R2 ) );
   }

   // rotation of a vector
   // overloaded multiplication of a vector by a rotation (using quaternions)
   friend Vector operator*( const Rotation& R, const Vector& a ) {

      quaternion q( to_quaternion( R ) );
      double w( q.w );
      Vector v( q.v );

      Vector b = 2. * ( v ^ a );
      return a + ( w * b ) + ( v ^ b );
   }

   // spherical linear interpolation on the unit sphere from u1 to u2
   friend Vector slerp( const Vector& u1, const Vector& u2, double theta, double t ) {

      assert( theta != 0 );
      assert( 0. <= t && t <= 1. );
      return ( sin( ( 1. - t ) * theta ) * u1 + sin( t * theta ) * u2 ) / sin( theta );
   }

   // spherical linear interpolation on the unit sphere from u1 to u2
   friend Vector slerp( const Vector& u1, const Vector& u2, double t ) {

      double theta = angle( u1, u2 );
      if ( theta == 0. ) return u1;
      assert( 0. <= t && t <= 1. );
      return ( sin( ( 1. - t ) * theta ) * u1 + sin( t * theta ) * u2 ) / sin( theta );
   }

   // access functions

   // return the unit axial vector
   friend Vector vec( const Rotation& R ) {   // return axial unit eigenvector

      return R._vec;
   }

   // return the rotation angle
   friend double ang( const Rotation& R ) {   // return rotation angle (rad)

      return R._ang;
   }

   // inverse rotation
   friend Rotation inverse( Rotation R ) {

      return Rotation( R._vec, -R._ang );
   }

   // conversion to quaternion
   friend quaternion to_quaternion( const Rotation& R ) {

      double a = 0.5 * R._ang;
      Vector u = R._vec;

      return quaternion( cos( a ), u * sin( a ) );
   }
```

```

    // conversion to rotation matrix
    friend matrix to_matrix( const Rotation& R ) {

       quaternion q( to_quaternion( R ) );
       double w = q.w;
       Vector v = q.v;
       double v1 = v[ X ], v2 = v[ Y ], v3 = v[ Z ];

       matrix A;
       A.a11 = 2. * ( w * w - 0.5 + v1 * v1 );   // 1st row, 1st col
       A.a12 = 2. * ( v1 * v2 - w * v3 );        // 1st row, 2nd col
       A.a13 = 2. * ( v1 * v3 + w * v2 );        // 1st row, 3rd col

       A.a21 = 2. * ( v1 * v2 + w * v3 );        // 2nd row, 1st col
       A.a22 = 2. * ( w * w - 0.5 + v2 * v2 );   // 2nd row, 2nd col
       A.a23 = 2. * ( v2 * v3 - w * v1 );        // 2nd row, 3rd col

       A.a31 = 2. * ( v1 * v3 - w * v2 );        // 3rd row, 1st col
       A.a32 = 2. * ( v2 * v3 + w * v1 );        // 3rd row, 2nd col
       A.a33 = 2. * ( w * w - 0.5 + v3 * v3 );   // 3rd row, 3rd col

       return A;
    }

    // factor rotation into a rotation sequence
    friend sequence factor( const Rotation& R, ORDER order ) {

       sequence s;
       return s.factor( to_quaternion( R ), order );
    }

    // factor matrix representation of rotation into a rotation sequence
    friend sequence factor( const matrix& A, ORDER order ) {

       sequence s;
       return s.factor( to_quaternion( Rotation( A ) ), order );
    }

    // overloaded stream operators

    // input a rotation
    friend std::istream& operator>>( std::istream& is, Rotation& R ) {

       std::cout << "Specify axis of rotation by entering an axial vector (need not be a unit vector)" << std::endl;
       is >> R._vec;
       std::cout << "Enter the angle of rotation (deg): ";
       is >> R._ang;

       R._vec = unit( R._vec );   // store the unit vector representing the axis
       R._ang = R._ang * D2R;     // store the rotation angle in radians
       return is;
    }

    // output a rotation
    friend std::ostream& operator<<( std::ostream& os, const Rotation& R ) {

       return os << R._vec << "\t" << R._ang * R2D;
    }

    // output a quaternion
    friend std::ostream& operator<<( std::ostream& os, const quaternion& q ) {

       return os << q.w << "\t" << q.v;
    }

    // output a matrix
    friend std::ostream& operator<<( std::ostream& os, const matrix& A ) {

       matrix B = set_precision( A );   // no more than 15 decimal digits of accuracy
       return os << B.a11 << "\t" << B.a12 << "\t" << B.a13 << std::endl
                 << B.a21 << "\t" << B.a22 << "\t" << B.a23 << std::endl
                 << B.a31 << "\t" << B.a32 << "\t" << B.a33;
    }

 public:

    // constructor from three angles (rad), phi_1, phi_2, phi_3 (in that order, left to right)
    // about three distinct principal body axes
    Rotation( double phi_1, double phi_2, double phi_3, ORDER order ) {

       double ang_1 = 0.5 * phi_1, c1 = cos( ang_1 ), s1 = sin( ang_1 );
       double ang_2 = 0.5 * phi_2, c2 = cos( ang_2 ), s2 = sin( ang_2 );
       double ang_3 = 0.5 * phi_3, c3 = cos( ang_3 ), s3 = sin( ang_3 );
       double w;
```

```
      Vector v;

      if ( order == ZYX ) {   // 1st about z-axis, 2nd about y-axis, 3rd about x-axis (Aerospace sequence)

         w = c1 * c2 * c3 + s1 * s2 * s3;
         v = Vector(  c1 * c2 * s3 - s1 * s2 * c3,
                      c1 * s2 * c3 + s1 * c2 * s3,
                     -c1 * s2 * s3 + s1 * c2 * c3 );
      }
      else if ( order == XYZ ) {   // 1st about x-axis, 2nd about y-axis, 3rd about z-axis (FATEPEN sequence)

         w = c1 * c2 * c3 - s1 * s2 * s3;
         v = Vector(  c1 * s2 * s3 + s1 * c2 * c3,
                      c1 * s2 * c3 - s1 * c2 * s3,
                      c1 * c2 * s3 + s1 * s2 * c3 );
      }
      else if ( order == YXZ ) {   // 1st about y-axis, 2nd about x-axis, 3rd about z-axis

         w = c1 * c2 * c3 + s1 * s2 * s3;
         v = Vector(  c1 * s2 * c3 + s1 * c2 * s3,
                     -c1 * s2 * s3 + s1 * c2 * c3,
                      c1 * c2 * s3 - s1 * s2 * c3 );
      }
      else if ( order == ZXY ) {   // 1st about z-axis, 2nd about x-axis, 3rd about y-axis

         w = c1 * c2 * c3 - s1 * s2 * s3;
         v = Vector(  c1 * s2 * c3 - s1 * c2 * s3,
                      c1 * c2 * s3 + s1 * s2 * c3,
                      c1 * s2 * s3 + s1 * c2 * c3 );
      }
      else if ( order == XZY ) {   // 1st about x-axis, 2nd about z-axis, 3rd about y-axis

         w = c1 * c2 * c3 + s1 * s2 * s3;
         v = Vector( -c1 * s2 * s3 + s1 * c2 * c3,
                      c1 * c2 * s3 - s1 * s2 * c3,
                      c1 * s2 * c3 + s1 * c2 * s3 );
      }
      else if ( order == YZX ) {   // 1st about y-axis, 2nd about z-axis, 3rd about x-axis

         w = c1 * c2 * c3 - s1 * s2 * s3;
         v = Vector(  c1 * c2 * s3 + s1 * s2 * c3,
                      c1 * s2 * s3 + s1 * c2 * c3,
                      c1 * s2 * c3 - s1 * c2 * s3 );
      }
      else if ( order == ZYZ ) {   // Euler sequence, 1st about z-axis, 2nd about y-axis, 3rd about z-axis

         w = c1 * c2 * c3 - s1 * c2 * s3;
         v = Vector(  c1 * s2 * s3 - s1 * s2 * c3,
                      c1 * s2 * c3 + s1 * s2 * s3,
                      c1 * c2 * s3 + s1 * c2 * c3 );
      }
      else if ( order == ZXZ ) {   // Euler sequence, 1st about z-axis, 2nd about x-axis, 3rd about z-axis

         w = c1 * c2 * c3 - s1 * c2 * s3;
         v = Vector(  c1 * s2 * c3 + s1 * s2 * s3,
                     -c1 * s2 * s3 + s1 * s2 * c3,
                      c1 * c2 * s3 + s1 * c2 * c3 );
      }
      else if ( order == YZY ) {   // Euler sequence, 1st about y-axis, 2nd about z-axis, 3rd about y-axis

         w = c1 * c2 * c3 - s1 * c2 * s3;
         v = Vector( -c1 * s2 * s3 + s1 * s2 * c3,
                      c1 * c2 * s3 + s1 * c2 * c3,
                      c1 * s2 * c3 + s1 * s2 * s3 );
      }
      else if ( order == YXY ) {   // Euler sequence, 1st about y-axis, 2nd about x-axis, 3rd about y-axis

         w = c1 * c2 * c3 - s1 * c2 * s3;
         v = Vector(  c1 * s2 * c3 + s1 * s2 * s3,
                      c1 * c2 * s3 + s1 * c2 * c3,
                      c1 * s2 * s3 - s1 * s2 * c3 );
      }
      else if ( order == XYX ) {   // Euler sequence, 1st about x-axis, 2nd about y-axis, 3rd about x-axis

         w = c1 * c2 * c3 - s1 * c2 * s3;
         v = Vector(  c1 * c2 * s3 + s1 * c2 * c3,
                      c1 * s2 * c3 + s1 * s2 * s3,
                     -c1 * s2 * s3 + s1 * s2 * c3 );
      }
      else if ( order == XZX ) {   // Euler sequence, 1st about x-axis, 2nd about z-axis, 3rd about x-axis

         w = c1 * c2 * c3 - s1 * c2 * s3;
         v = Vector(  c1 * c2 * s3 + s1 * c2 * c3,
                      c1 * s2 * s3 - s1 * s2 * c3,
                      c1 * s2 * c3 + s1 * s2 * s3 );
```

```
      }
      else {

         std::cerr << "ERROR in Rotation: invalid order: " << order << std::endl;
         exit( EXIT_FAILURE );
      }
      if ( w >= 1. || v == 0. ) {
         _ang = DEFAULT_ROTATION_ANGLE;
         _vec = DEFAULT_UNIT_VECTOR;
      }
      else {
         _ang = 2. * acos( w );
         _vec = v / sqrt( 1. - w * w );
      }
      _set_angle();   // angle in the range [-M_PI, M_PI]
   }

   // constructor from a rotation sequence
   Rotation( const sequence& s, ORDER order ) {

      Rotation R( s.first, s.second, s.third, order );

      _vec = vec( R );     // set the axial vector
      _ang = ang( R );     // set the rotation angle
      _set_angle();   // angle in the range [-M_PI, M_PI]
   }

   // constructor from an axial vector and rotation angle (rad)
   Rotation( const Vector& v, double a ) : _vec( v ), _ang( a ) {

      _vec = _vec.unit();   // store the unit vector representing the axis
      _set_angle();    // angle in the range [-M_PI, M_PI]
   }

   // constructor using sphericalCoord (of axial vector) and rotation angle (rad)
   Rotation( rng::sphericalCoord s, double ang ) {

      _vec = Vector( 1., s.theta, s.phi, POLAR );   // unit vector
      _ang = ang;
      _set_angle();   // angle in the range [-M_PI, M_PI]
   }

   // constructor from the cross product of two vectors
   // generate the rotation that, when applied to vector a, will result in vector b
   Rotation( const Vector& a, const Vector& b ) {

      _vec = unit( a ^ b );   // unit vector
      double s = a.unit() * b.unit();
      if ( s >= 1. )
         _ang = 0.;
      else if ( s <= -1. )
         _ang= M_PI;
      else
         _ang = acos( s );

      _set_angle();   // angle in the range [-M_PI, M_PI]
   }

   // constructor from unit quaternion
   Rotation( const quaternion& q ) {

      double w = q.w;
      Vector v = q.v;

      if ( w >= 1. || v == 0. ) {
         _ang = DEFAULT_ROTATION_ANGLE;
         _vec = DEFAULT_UNIT_VECTOR;
      }
      else {
         double n = sqrt( w * w + v * v );   // need to insure it's a unit quaternion
         w /= n;
         v /= n;
         _ang = 2. * acos( w );
         _vec = v / sqrt( 1. - w * w );
      }
      _set_angle();   // angle in the range [-M_PI, M_PI]
   }

   // constructor from rotation matrix
   Rotation( const matrix& A ) {

      _vec = ( A.a32 - A.a23 ) * Vector( 1., 0., 0. ) +
             ( A.a13 - A.a31 ) * Vector( 0., 1., 0. ) +
             ( A.a21 - A.a12 ) * Vector( 0., 0., 1. );
      if ( _vec == 0. ) {   // then it must be the identity matrix
```

```
         _vec = DEFAULT_UNIT_VECTOR;
         _ang = DEFAULT_ROTATION_ANGLE;
      }
      else {
         _vec = _vec.unit();
         _ang = acos( 0.5 * ( A.a11 + A.a22 + A.a33 - 1. ) );
      }
      _set_angle();   // angle in the range [-M_PI, M_PI]
   }

   // constructor from two sets of three vectors, where the pair must be related by a pure rotation
   // returns the rotation that will take a1 to b1, a2 to b2, and a3 to b3
   // Ref: Micheals, R. J. and Boult, T. E., "Increasing Robustness in Self-Localization and Pose Estimation," online paper.

   Rotation( const Vector& a1, const Vector& a2, const Vector& a3,      // initial vectors
             const Vector& b1, const Vector& b2, const Vector& b3 ) {   // rotated vectors

      assert( det( a1, a2, a3 ) != 0. && det( b1, b2, b3 ) != 0. );   // all vectors must be nonzero
      assert( fabs( det( a1, a2, a3 ) - det( b1, b2, b3 ) ) < 0.001 );   // if it doesn't preserve volume, it's not a pure rotation

      if ( det( a1, a2, a3 ) == 1 ) {   // these are basis vectors so use simpler method to construct the rotation

         matrix A;
         A.a11 = b1 * a1; A.a12 = b2 * a1; A.a13 = b3 * a1;
         A.a21 = b1 * a2; A.a22 = b2 * a2; A.a23 = b3 * a2;
         A.a31 = b1 * a3; A.a32 = b2 * a3; A.a33 = b3 * a3;

         _vec = ( A.a32 - A.a23 ) * Vector( 1., 0., 0. ) +
                ( A.a13 - A.a31 ) * Vector( 0., 1., 0. ) +
                ( A.a21 - A.a12 ) * Vector( 0., 0., 1. );
         if ( _vec == 0. ) {   // then it must be the identity matrix
            _vec = DEFAULT_UNIT_VECTOR;
            _ang = DEFAULT_ROTATION_ANGLE;
         }
         else {
            _vec = _vec.unit();
            _ang = acos( 0.5 * ( A.a11 + A.a22 + A.a33 - 1. ) );
         }
         _set_angle();   // angle in the range [-M_PI, M_PI]
      }
      else {   // use R. J. Micheals' closed-form solution to the absolute orientation problem

         Vector c1, c2, c3;
         double aaa, baa, aba, aab, caa, aca, aac;
         double q02, q0, q0q1, q1, q0q2, q2, q0q3, q3;

         aaa = det( a1, a2, a3 );
         baa = det( b1, a2, a3 );
         aba = det( a1, b2, a3 );
         aab = det( a1, a2, b3 );

         q02 = fabs( ( aaa + baa + aba + aab ) / ( 4. * aaa ) );
         q0 = sqrt( q02 );

         c1 = Vector( 0., b1[Z], -b1[Y] );
         c2 = Vector( 0., b2[Z], -b2[Y] );
         c3 = Vector( 0., b3[Z], -b3[Y] );

         caa = det( c1, a2, a3 );
         aca = det( a1, c2, a3 );
         aac = det( a1, a2, c3 );

         q0q1 = ( caa + aca + aac ) / ( 4. * aaa );
         q1 = q0q1 / q0;

         c1 = Vector( -b1[Z], 0., b1[X] );
         c2 = Vector( -b2[Z], 0., b2[X] );
         c3 = Vector( -b3[Z], 0., b3[X] );

         caa = det( c1, a2, a3 );
         aca = det( a1, c2, a3 );
         aac = det( a1, a2, c3 );

         q0q2 = ( caa + aca + aac ) / ( 4. * aaa );
         q2 = q0q2 / q0;

         c1 = Vector( b1[Y], -b1[X], 0. );
         c2 = Vector( b2[Y], -b2[X], 0. );
         c3 = Vector( b3[Y], -b3[X], 0. );

         caa = det( c1, a2, a3 );
         aca = det( a1, c2, a3 );
         aac = det( a1, a2, c3 );
```

```
         q0q3 = ( caa + aca + aac ) / ( 4. * aaa );
         q3 = q0q3 / q0;

         // no need to normalize since constructed to be unit quaternions
         double w( q0 );
         Vector v( q1, q2, q3 );

         if ( w >= 1. || v == 0. ) {
            _ang = DEFAULT_ROTATION_ANGLE;
            _vec = DEFAULT_UNIT_VECTOR;
         }
         else {
            _ang = 2. * acos( w );
            _vec = v / sqrt( 1. - w * w );
         }
         _set_angle();   // angle in the range [-M_PI, M_PI]
      }
   }

   // constructor for a uniform random rotation, uniformly distributed over the unit sphere,
   // by fast generation of random quaternions, uniformly-distributed over the 4D unit sphere
   // Ref: Shoemake, K., "Uniform Random Rotations," Graphic Gems III, September, 1991.
   Rotation( rng::Random& rng ) {   // random rotation in canonical form

      double s   = rng.uniform( 0., 1. );
      double s1  = sqrt( 1. - s );
      double th1 = rng.uniform( 0., TWO_PI );
      double x   = s1 * sin( th1 );
      double y   = s1 * cos( th1 );
      double s2  = sqrt( s );
      double th2 = rng.uniform( 0., TWO_PI );
      double z   = s2 * sin( th2 );
      double w   = s2 * cos( th2 );
      Vector v( x, y, z );

      if ( w >= 1. || v == 0. ) {
         _ang = DEFAULT_ROTATION_ANGLE;
         _vec = DEFAULT_UNIT_VECTOR;
      }
      else {
         _ang = 2. * acos( w );
         _vec = v / sqrt( 1. - w * w );
      }
      _set_angle();   // angle in the range [-M_PI, M_PI]
   }

   // default constructor
   Rotation( void ) {

      _vec = DEFAULT_UNIT_VECTOR;
      _ang = DEFAULT_ROTATION_ANGLE;
   }

   // default destructor
  ~Rotation( void ) {
   }

   // copy constructor
   Rotation( const Rotation& r ) : _vec( r._vec ), _ang( r._ang ) {

      _set_angle();   // angle in the range [-M_PI, M_PI]
   }

   // overloaded assignment operator
   Rotation& operator=( const Rotation& R ) {

      if ( this != &R ) {
         _vec = R._vec;
         _ang = R._ang;
         _set_angle();   // angle in the range [-M_PI, M_PI]
      }
      return *this;
   }

   // conversion operator to return the eigenvector vec
   operator Vector( void ) const {

      return _vec;
   }

   // conversion operator to return the angle of rotation about the eigenvector
   operator double( void ) const {

      return _ang;
```

```
    }

    // overloaded arithmetic operators

    // inverse rotation
    Rotation operator-( void ) {

       return Rotation( -_vec, _ang );
    }

    // triple scalar product, same as a * ( b ^ c )
    inline double det( const Vector& a, const Vector& b, const Vector& c ) {

       return a[X] * ( b[Y] * c[Z] - b[Z] * c[Y] ) +
              a[Y] * ( b[Z] * c[X] - b[X] * c[Z] ) +
              a[Z] * ( b[X] * c[Y] - b[Y] * c[X] );
    }

private:

    inline void _set_angle( void ) {   // always choose the smaller of the two angles

       if ( _ang > +TWO_PI ) _ang -= TWO_PI;
       if ( _ang < -TWO_PI ) _ang += TWO_PI;
       if ( _ang > M_PI ) {
          _ang = TWO_PI - _ang;
          _vec = -_vec;
       }
       else if ( _ang < -M_PI ) {
          _ang = TWO_PI + _ang;
          _vec = -_vec;
       }
    }

    Vector _vec;   // unit eigenvector representing the axis of rotation
    double _ang;   // angle of rotation (rad) falls in the range [-M_PI, M_PI]
};

// declaration of friends
    quaternion operator*( const quaternion& q1, const quaternion& q2 );           // product of two quaternions
    matrix operator*( const matrix& A, const matrix& B );                         // product of two matrices, first B, then A
    matrix transpose( const matrix& A );                                          // transpose of a matrix
    matrix inverse( const matrix& A );                                            // inverse of a matrix
    matrix set_precision( const matrix& A );                                      // returns a matrix with no more than 15 decimal digit accuracy
    double tr( const matrix& A );                                                 // trace of a matrix
    double det( const matrix& A );                                                // determinant of a matrix
    Vector eigenvector( const matrix& A );                                        // eigenvector of a rotation matrix
    double angle( const matrix& A );                                              // angle of rotation (rad)
    Rotation operator*( const Rotation& R1, const Rotation& R2 );                 // successive rotations, first right, then left
    Vector operator*( const Rotation& R, const Vector& a );                       // rotation of a vector
    Vector slerp( const Vector& u1, const Vector& u2, double t );                  // spherical linear interpolation on the unit sphere
    Vector slerp( const Vector& u1, const Vector& u2, double theta, double t );    // slerp, given the angle between the vectors
    Vector vec( const Rotation& R );                                              // return axial unit eigenvector
    double ang( const Rotation& R );                                              // return rotation angle (rad)
    Rotation inverse( Rotation R );                                               // inverse rotation
    quaternion to_quaternion( const Rotation& R );                                // convert rotation to a quaternion
    matrix to_matrix( const Rotation& R );                                        // convert a rotation to a rotation matrix
    sequence factor( const Rotation& R, ORDER order );                            // factor a rotation into a rotation sequence
    sequence factor( const matrix& A, ORDER order );                              // factor a rotation matrix into a rotation sequence
    std::istream& operator>>( std::istream& is, Rotation& R );                    // input rotation
    std::ostream& operator<<( std::ostream& os, const Rotation& R );              // output a rotation
    std::ostream& operator<<( std::ostream& os, const quaternion& q );            // output a quaternion
    std::ostream& operator<<( std::ostream& os, const matrix& A );                // output a rotation matrix

} // vector algebra namespace
#endif
```

The Rotation class is also enclosed in a `va` namespace, so that Rotations are declared by `va::Rotation` or by the declaration `using namespace va`. Table B-1 provides a reference sheet for basic usage.

Table B-1. Rotation: A C++ class for 3-dimensional rotations—reference sheet

Operation	Mathematical notation	Rotation class
Definition[a]	Let R be an unspecified rotation.	`Rotation R;`
	Let R be a rotation specified by yaw, pitch, and roll.[b]	`Rotation R(y,p,r,ZYX);`
	Let R be the rotation specified by three angles, ϕ_1, ϕ_2, ϕ_3 applied in the order x-y-z.	`Rotation b(ph1,ph2,ph3,XYZ);`
	Let $R_{\hat{\boldsymbol{a}}}(\alpha)$ be the rotation about the vector $\boldsymbol{a}$ through the angle α.	`Vector a;` `Rotation R(a,alpha);`
	Let R be the rotation about the direction specified by the angles (θ, ϕ) through the angle α.	`pair<double,double> p(th,ph);` `Rotation R(p,alpha);`
	Let R be the rotation specified by the vector cross product $\boldsymbol{a} \times \boldsymbol{b}$.	`Vector a, b;` `Rotation R(a,b);`
	Let R be the rotation that maps the set of linearly independent vectors $\boldsymbol{a}_i$ to the set $\boldsymbol{b}_i$, where $i = 1, 2, 3$.	`Vector a1,a2,a3,b1,b2,b3;` `Rotation R(a1,a2,a3,b1,b2,b3);`
	Let R be the rotation specified by the (unit) quaternion q.[c]	`quaternion q;` `Rotation R(q);`
	Let R be the rotation specified by the 3×3 rotation matrix A_{ij}.[d]	`matrix A;` `Rotation R(A);`
	Let R be a random rotation, designed to randomly orient any vector uniformly over the unit sphere.	`rng::Random rng;` `R(rng);`
Input a rotation R	*n/a*	`cin >> R;`
Output the rotation R	*n/a*	`cout << R;`
Assign one rotation to another	Let $R_2 = R_1$ *or* $R_2 \Leftarrow R_1$	`R2 = R1;` *or* `R2(R1);`
Product of two successive rotations[e]	$R_2 R_1$	`R2 * R1;`
Rotation of a vector $\boldsymbol{a}$	$R\,\boldsymbol{a}$	`R * a;`
Inverse rotation	R^{-1}	`inverse(R);` *or* `-R;`
Convert a rotation to a quaternion	If $R_{\hat{\boldsymbol{u}}}(\theta)$ is the rotation, then $q = \cos(\theta/2) + \hat{\boldsymbol{u}} \sin(\theta/2)$.	`to_quaternion(R);`
Convert a rotation to a 3×3 matrix	*See description on next page.*	`to_matrix(R);`
Factor a rotation into a rotation sequence	*See description on next page.*	`sequence s = factor(R,ZYX);`[f]
Unit vector along the axis of rotation	Unit vector $\hat{\boldsymbol{u}}$ in the rotation $R_{\hat{\boldsymbol{u}}}(\theta)$	`Vector(R);` *or* `vec(R);`
Rotation angle	Angle θ in the rotation $R_{\hat{\boldsymbol{u}}}(\theta)$	`double(R);` *or* `ang(R);`

[a] A rotation is represented in the Rotation class by the pair $(\hat{\boldsymbol{u}}, \theta)$, where $\hat{\boldsymbol{u}}$ is the unit vector along the axis of rotation, and θ is the counterclockwise rotation angle.

[b] The order is significant: first yaw is applied as a counterclockwise (CCW) rotation about the z-axis, then pitch is applied as a CCW rotation about the y'-axis, and finally, roll is applied as a CCW rotation about the x''-axis. The coordinate system is constructed from the local tangent plane in which the z-axis points toward earth center, the x-axis points along the direction of travel, and the y-axis points to the right, to form a right-handed coordinate system. The particular order is specified by using `ZYX`. There are a total of 12 possible orderings available to the user, 6 of them have distinct principal rotation axes: `XYZ, XZY, YXZ, YZX, ZXY, ZYX`; and 6 have repeated principal rotation axes: `XYX, XZX, YXY, YZY, ZXZ, ZYZ`.

Convert rotation to a 3×3 matrix: First we convert the rotation $R_{\hat{\boldsymbol{u}}}(\theta)$ into the unit quaternion, via $q = \cos(\theta/2) + \hat{\boldsymbol{u}} \sin(\theta/2)$, and set $w = \cos(\theta/2)$, the scalar part, and $\boldsymbol{v} = \hat{\boldsymbol{u}} \sin(\theta/2)$, the vector part. Then the rotation matrix is

$$\begin{bmatrix} 2w^2 - 1 + 2v_1^2 & 2v_1v_2 - 2wv_3 & 2v_1v_3 + 2wv_2 \\ 2v_1v_2 + 2wv_3 & 2w^2 - 1 + 2v_2^2 & 2v_2v_3 - 2wv_1 \\ 2v_1v_3 - 2wv_2 & 2v_2v_3 + 2wv_1 & 2w^2 - 1 + 2v_3^2 \end{bmatrix}.$$

Factor a rotation into a rotation sequence: First we convert the rotation $R_{\hat{\boldsymbol{u}}}(\theta)$ into the unit quaternion, via $q = \cos(\theta/2) + \hat{\boldsymbol{u}} \sin(\theta/2)$, and set $w = \cos(\theta/2)$, the scalar part, and $\boldsymbol{v} = \hat{\boldsymbol{u}} \sin(\theta/2)$, the vector part. Next, let $p_0 = w$, $p_1 = v_1$, $p_2 = v_2$, $p_3 = v_3$ and set $A = p_0p_1 + p_2p_3$, $B = p_2^2 - p_0^2$, $D = p_1^2 - p_3^2$. Then $\phi_3 = \tan^{-1}(-2A/(B + D))$ is the third angle, which is *roll* about the x-axis in this case. Now set $c_0 = \cos(\phi_3/2)$, $c_1 = \sin(\phi_3/2)$, $q_0 = p_0c_0 + p_1c_1$, $q_2 = p_2c_0 - p_3c_1$, and $q_3 = p_3c_0 + p_2c_1$. Then $\phi_1 = 2\tan^{-1}(q_3/q_0)$ is the first angle, which is *yaw* about the z-axis, and $\phi_2 = 2\tan^{-1}(q_2/q_0)$ is the second angle, which is *pitch* about the y-axis.

---

[c] A quaternion is defined in the Rotation class as follows:

```
struct quaternion {
double w; // scalar part
Vector v; // vector part
};
```

A *unit* quaternion requires that $w^2 + \|\boldsymbol{v}\|^2 = 1$.

[d] A matrix is defined in the Rotation class as follows:

```
struct matrix {
double a11, a12, a13; // 1st row
double a21, a22, a23; // 2nd row
double a31, a32, a33; // 3rd row
};
```

To qualify as a rotation, the 3 matrix A must satisfy the 2 conditions: $A^\dagger = A^{-1}$ and $\det A = 1$.

[e] In general, rotations do not commute, i.e. $R_1R_2 \neq R_2R_1$, so the order is significant and goes from right to left.

[f] A (rotation) sequence is defined in the Rotation class as follows:

```
struct sequence {
double first; // 1st rotation (rad) to apply to body axis
double second; // 2nd rotation (rad) to apply to body axis
double third; // 3rd rotation (rad) to apply to body axis
};
```

The order these are applied is always left to right: `first, second, third`. How they get applied is specified by using one of the following, which is applied left to right: `ZYX, XYZ, XZY, YZX, YXZ, ZYX, ZYZ, ZXZ, YZY, YXY, XYX, XZX`. For example, the order `XYZ` would apply `first` to rotation about the x-axis, `second` to rotation about the y-axis, and `third` to rotation about the z-axis.

INTENTIONALLY LEFT BLANK.

Appendix C. Quaternion Algebra and Vector Rotations

C-1. Quaternion Multiplication

Starting with the multiplication rule

$$\hat{\imath}^2 = \hat{\jmath}^2 = \hat{\mathbf{k}}^2 = \hat{\imath}\,\hat{\jmath}\,\hat{\mathbf{k}} = -1, \tag{C-1}$$

it then follows that

$$\hat{\imath}\hat{\jmath} = -\hat{\jmath}\hat{\imath} = \hat{\mathbf{k}}, \quad \hat{\jmath}\hat{\mathbf{k}} = -\hat{\mathbf{k}}\hat{\jmath} = \hat{\imath}, \quad \text{and} \quad \hat{\mathbf{k}}\hat{\imath} = -\hat{\imath}\hat{\mathbf{k}} = \hat{\jmath}. \tag{C-2}$$

These rules are then sufficient to establish any other multiplication. Thus, let

$$\begin{aligned} q_1 &= w_1 + \boldsymbol{v}_1 = w_1 + \hat{\imath}x_1 + \hat{\jmath}y_1 + \hat{\mathbf{k}}z_1 \\ q_2 &= w_2 + \boldsymbol{v}_2 = w_2 + \hat{\imath}x_2 + \hat{\jmath}y_2 + \hat{\mathbf{k}}z_2 \end{aligned}$$

be 2 quaternions. Unlike vectors, where there are 2 different products—the scalar product and the vector product—in the case of quaternions there is only one product, as follows:

$$\begin{aligned} q_1q_2 &= (w_1 + \hat{\imath}x_1 + \hat{\jmath}y_1 + \hat{\mathbf{k}}z_1)(w_2 + \hat{\imath}x_2 + \hat{\jmath}y_2 + \hat{\mathbf{k}}z_2) \\ &= (w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2) \\ &+ \hat{\imath}\,(w_1x_2 + w_2x_1 + y_1z_2 - z_1y_2) \\ &+ \hat{\jmath}\,(w_1y_2 + w_2y_1 + z_1x_2 - x_1z_2) \\ &+ \hat{\mathbf{k}}\,(w_1z_2 + w_2z_1 + x_1y_2 - y_1x_2) \\ &= w_1w_2 - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2 + w_1\boldsymbol{v}_2 + w_2\boldsymbol{v}_1 + \boldsymbol{v}_1 \times \boldsymbol{v}_2. \end{aligned} \tag{C-3}$$

Thus, if we represent a quaternion as an ordered pair, $q = (w, \boldsymbol{v})$, of a scalar and a vector, then

$$(w_1, \boldsymbol{v}_1)(w_2, \boldsymbol{v}_2) = (w_1w_2 - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2,\; w_1\boldsymbol{v}_2 + w_2\boldsymbol{v}_1 + \boldsymbol{v}_1 \times \boldsymbol{v}_2). \tag{C-4}$$

The scalar part of the product is

$$w_1w_2 - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2$$

and the vector part is

$$w_1\boldsymbol{v}_2 + w_2\boldsymbol{v}_1 + \boldsymbol{v}_1 \times \boldsymbol{v}_2.$$

C-2. Quaternion Division

Let $q = w + \boldsymbol{v}$ be a unit quaternion, in the sense that $w^2 + \|\boldsymbol{v}\|^2 = 1$. Then $q^{-1} = w - \boldsymbol{v}$ is the *inverse*, since

$$\begin{aligned} qq^{-1} &= (w, \boldsymbol{v})(w, -\boldsymbol{v}) \\ &= (w^2 - \boldsymbol{v} \cdot (-\boldsymbol{v}),\ w(-\boldsymbol{v}) + w\boldsymbol{v} + \boldsymbol{v} \times (-\boldsymbol{v})) \\ &= (w^2 + \|\boldsymbol{v}\|^2,\ \boldsymbol{0}) \\ &= 1. \end{aligned} \tag{C-5}$$

Thus, the inverse of a unit quaternion is the quaternion with a negative vector part. In effect, this serves to define quaternion division.*

C-3. Rotation of a Vector

Let $\boldsymbol{a}$ be an arbitrary vector, let $\hat{\boldsymbol{u}}$ be a unit vector along the axis of rotation, and let θ be the angle of rotation. The (unit) quaternion that represents the rotation is given by

$$q = \left(\cos\frac{\theta}{2}, \hat{\boldsymbol{u}} \sin\frac{\theta}{2}\right) \equiv (w, \boldsymbol{v}). \tag{C-6}$$

Then the rotated vector, $\boldsymbol{a}'$ is given by

$$\begin{aligned} \boldsymbol{a}' &= q\boldsymbol{a}q^{-1} \\ &= (w, \boldsymbol{v})(0, \boldsymbol{a})(w, -\boldsymbol{v}) \\ &= (w, \boldsymbol{v})(\boldsymbol{a} \cdot \boldsymbol{v}, w\boldsymbol{a} - \boldsymbol{a} \times \boldsymbol{v}) \\ &= (w\boldsymbol{a} \cdot \boldsymbol{v} - \boldsymbol{v} \cdot (w\boldsymbol{a} - \boldsymbol{a} \times \boldsymbol{v}), w(w\boldsymbol{a} - \boldsymbol{a} \times \boldsymbol{v}) + (\boldsymbol{a} \cdot \boldsymbol{v})\boldsymbol{v} + \boldsymbol{v} \times (w\boldsymbol{a} - \boldsymbol{a} \times \boldsymbol{v})) \\ &= (0, w^2\boldsymbol{a} + w\boldsymbol{v} \times \boldsymbol{a} + (\boldsymbol{a} \cdot \boldsymbol{v})\boldsymbol{v} + w\boldsymbol{v} \times \boldsymbol{a} + \boldsymbol{v} \times (\boldsymbol{v} \times \boldsymbol{a})) \\ &= (0, w^2\boldsymbol{a} + v^2\boldsymbol{a} - v^2\boldsymbol{a} + (\boldsymbol{a} \cdot \boldsymbol{v})\boldsymbol{v} + 2w\boldsymbol{v} \times \boldsymbol{a} + \boldsymbol{v} \times (\boldsymbol{v} \times \boldsymbol{a})) \\ &= (0, \boldsymbol{a} + 2w\boldsymbol{v} \times \boldsymbol{a} + 2\boldsymbol{v} \times (\boldsymbol{v} \times \boldsymbol{a})), \end{aligned} \tag{C-7}$$

where we used the fact that $w^2 + v^2 = 1$ and $(\boldsymbol{a} \cdot \boldsymbol{v})\boldsymbol{v} - v^2\boldsymbol{a} = \boldsymbol{v} \times (\boldsymbol{v} \times \boldsymbol{a})$. Therefore,

*Quaternions form what is known as a *division algebra*, meaning that every non-zero quaternion has a multiplicative inverse. Vectors by themselves form an algebra but without division. For an interesting discussion of the relative merits of Hamilton's quaternions and Gibbs' vectors, see Chappell JM, Iqbal A, Hartnett JG, Abbott D. The vector algebra war: a historical perspective. Proc IEEE. 2016;4:1997–2004.

the rotated vector is given by

$$\boxed{\boldsymbol{a}' = \boldsymbol{a} + 2w\boldsymbol{v} \times \boldsymbol{a} + 2\boldsymbol{v} \times (\boldsymbol{v} \times \boldsymbol{a})}. \tag{C-8}$$

Using $w = \cos\theta/2$ and $\boldsymbol{v} = \hat{\boldsymbol{u}} \sin\theta/2$, we also have

$$\boxed{\boldsymbol{a}' = \boldsymbol{a} + \hat{\boldsymbol{u}} \times \boldsymbol{a}\, \sin\theta + \hat{\boldsymbol{u}} \times (\hat{\boldsymbol{u}} \times \boldsymbol{a})\,(1 - \cos\theta)}, \tag{C-9}$$

where we made use of the half-angle formulas $2\cos(\theta/2)\sin(\theta/2) = \sin\theta$ and $2\sin^2(\theta/2) = 1 - \cos\theta$.

Since this is such a fundamental formula, let us derive it in another way. For an arbitrary vector $\boldsymbol{a}$, we can always write

$$\boldsymbol{a} = \boldsymbol{a} - (\boldsymbol{a} \cdot \hat{\boldsymbol{u}})\hat{\boldsymbol{u}} + (\boldsymbol{a} \cdot \hat{\boldsymbol{u}})\hat{\boldsymbol{u}}, \tag{C-10}$$

where the third term on the right is the component of $\boldsymbol{a}$ that is parallel to $\hat{\boldsymbol{u}}$ and so will remain unchanged after a rotation about $\hat{\boldsymbol{u}}$. The first 2 terms form the component of $\boldsymbol{a}$ that is perpendicular to $\hat{\boldsymbol{u}}$ and will be rotated into

$$[\boldsymbol{a} - (\boldsymbol{a} \cdot \hat{\boldsymbol{u}})\hat{\boldsymbol{u}}] \cos\theta + \hat{\boldsymbol{u}} \times [\boldsymbol{a} - (\boldsymbol{a} \cdot \hat{\boldsymbol{u}})\hat{\boldsymbol{u}}] \sin\theta. \tag{C-11}$$

Hence,

$$\begin{aligned} \boldsymbol{a}' = R\boldsymbol{a} &= [\boldsymbol{a} - (\boldsymbol{a} \cdot \hat{\boldsymbol{u}})\hat{\boldsymbol{u}}] \cos\theta + \hat{\boldsymbol{u}} \times [\boldsymbol{a} - (\boldsymbol{a} \cdot \hat{\boldsymbol{u}})\hat{\boldsymbol{u}}] \sin\theta + +(\boldsymbol{a} \cdot \hat{\boldsymbol{u}})\hat{\boldsymbol{u}} \\ &= [\boldsymbol{a} - (\boldsymbol{a} \cdot \hat{\boldsymbol{u}})\hat{\boldsymbol{u}}] \cos\theta + \hat{\boldsymbol{u}} \times \boldsymbol{a} \sin\theta + (\boldsymbol{a} \cdot \hat{\boldsymbol{u}})\hat{\boldsymbol{u}}. \end{aligned} \tag{C-12}$$

Now,

$$\hat{\boldsymbol{u}} \times (\boldsymbol{a} \times \hat{\boldsymbol{u}}) = \boldsymbol{a}(\hat{\boldsymbol{u}} \cdot \hat{\boldsymbol{u}}) - \hat{\boldsymbol{u}}(\hat{\boldsymbol{u}} \cdot \boldsymbol{a}) = \boldsymbol{a} - (\boldsymbol{a} \cdot \hat{\boldsymbol{u}})\hat{\boldsymbol{u}},$$

and therefore

$$\begin{aligned} \boldsymbol{a}' &= \hat{\boldsymbol{u}} \times (\boldsymbol{a} \times \hat{\boldsymbol{u}}) \cos\theta + \hat{\boldsymbol{u}} \times \boldsymbol{a} \sin\theta + \boldsymbol{a} - \hat{\boldsymbol{u}} \times (\boldsymbol{a} \times \hat{\boldsymbol{u}}) \\ &= -\hat{\boldsymbol{u}} \times (\hat{\boldsymbol{u}} \times \boldsymbol{a}) \cos\theta + \hat{\boldsymbol{u}} \times \boldsymbol{a} \sin\theta + \boldsymbol{a} + \hat{\boldsymbol{u}} \times (\hat{\boldsymbol{u}} \times \boldsymbol{a}) \\ &= \boldsymbol{a} + \hat{\boldsymbol{u}} \times \boldsymbol{a}\, \sin\theta + \hat{\boldsymbol{u}} \times (\hat{\boldsymbol{u}} \times \boldsymbol{a})\,(1 - \cos\theta). \end{aligned} \tag{C-13}$$

Appendix D. Fundamental Theorem of Rotation Sequences

Fundamental Theorem of Rotation Sequences: A rotation sequence about body axes is equivalent to the same rotation sequence applied in reverse order about fixed axes.

The conventional way of performing a rotation sequence is to account for the transformation of the body axes of the object we are rotating. For example, if we wanted to first perform pitch about the x-axis, followed by yaw about the y-axis, and ending with roll about the z-axis, then the rotation, applied right to left, is

$$R = R_{\hat{\mathbf{k}}''}(\phi_r) R_{\hat{\mathbf{j}}'}(\phi_y) R_{\hat{\mathbf{\imath}}}(\phi_p), \tag{D-1}$$

where $\hat{\mathbf{j}}' = R_{\hat{\mathbf{\imath}}}(\phi_p)\,\hat{\mathbf{j}}$, $\hat{\mathbf{k}}' = R_{\hat{\mathbf{\imath}}}(\phi_p)\,\hat{\mathbf{k}}$, and $\hat{\mathbf{k}}'' = R_{\hat{\mathbf{j}}'}(\phi_y)\,\hat{\mathbf{k}}' = R_{\hat{\mathbf{j}}'}(\phi_y) R_{\hat{\mathbf{\imath}}}(\phi_p)\,\hat{\mathbf{k}}$. But it is a fundamental result of rotation sequences that you get the same result by applying the rotation sequence in reverse order about fixed axes. That is,

$$\boxed{R = R_{\hat{\mathbf{k}}''}(\phi_r) R_{\hat{\mathbf{j}}'}(\phi_y) R_{\hat{\mathbf{\imath}}}(\phi_p) = R_{\hat{\mathbf{\imath}}}(\phi_p) R_{\hat{\mathbf{j}}}(\phi_y) R_{\hat{\mathbf{k}}}(\phi_r)}, \tag{D-2}$$

which is simpler and more efficient. This can be proved with quaternions as follows. We use the notation,

$$q_{\hat{\mathbf{u}}}(\phi) = \cos\frac{\phi}{2} + \hat{\boldsymbol{u}}\sin\frac{\phi}{2} \tag{D-3}$$

for the unit quaternion that represents a counterclockwise rotation of ϕ radians about the unit vector $\hat{\boldsymbol{u}}$. Then,

$$R_1 = q_{\hat{\mathbf{e}}_1}(\phi_1). \tag{D-4}$$

$$R_2 = q_{\hat{\mathbf{e}}'_2}(\phi_2) = \cos\frac{\phi_2}{2} + \hat{\mathbf{e}}'_2\sin\frac{\phi_2}{2}, \tag{D-5}$$

where

$$\hat{\boldsymbol{e}}'_2 = q_{\hat{\mathbf{e}}_1}(\phi_1)\hat{\boldsymbol{e}}_2 q^{-1}_{\hat{\mathbf{e}}_1}(\phi_1), \tag{D-6}$$

so that

$$\begin{aligned} q_{\hat{\mathbf{e}}'_2}(\phi_2) &= \cos\frac{\phi_2}{2} + \hat{\boldsymbol{e}}'_2\sin\frac{\phi_2}{2} \\ &= \cos\frac{\phi_2}{2} + q_{\hat{\mathbf{e}}_1}(\phi_1)\hat{\boldsymbol{e}}_2 q^{-1}_{\hat{\mathbf{e}}_1}(\phi_1)\sin\frac{\phi_2}{2} \\ &= q_{\hat{\mathbf{e}}_1}(\phi_1)\left(\cos\frac{\phi_2}{2} + \hat{\boldsymbol{e}}_2\sin\frac{\phi_2}{2}\right) q^{-1}_{\hat{\mathbf{e}}_1}(\phi_1) \\ &= q_{\hat{\mathbf{e}}_1}(\phi_1) q_{\hat{\mathbf{e}}_2}(\phi_2) q^{-1}_{\hat{\mathbf{e}}_1}(\phi_1). \end{aligned} \tag{D-7}$$

And

$$R_3 = q_{\hat{e}_3''}(\phi_3) = \cos\frac{\phi_3}{2} + \hat{\boldsymbol{e}}_3'' \sin\frac{\phi_3}{2}, \tag{D-8}$$

where

$$\begin{aligned}
\hat{\boldsymbol{e}}_3'' &= q_{\hat{e}_2'}(\phi_2)\hat{\boldsymbol{e}}_3' q_{\hat{e}_2'}^{-1}(\phi_2) \\
&= q_{\hat{e}_2'}(\phi_2) q_{\hat{e}_1}(\phi_1)\hat{\boldsymbol{e}}_3 q_{\hat{e}_1}^{-1}(\phi_1) q_{\hat{e}_2'}^{-1}(\phi_2) \\
&= [q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1)] q_{\hat{e}_1}(\phi_1)\hat{\boldsymbol{e}}_3 q_{\hat{e}_1}^{-1}(\phi_1)[q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1)]^{-1} \\
&= q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1) q_{\hat{e}_1}(\phi_1)\hat{\boldsymbol{e}}_3 q_{\hat{e}_1}^{-1}(\phi_1) q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1) \\
&= q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2)\hat{\boldsymbol{e}}_3 q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1),
\end{aligned} \tag{D-9}$$

so that

$$\begin{aligned}
q_{\hat{e}_3''}(\phi_3) &= \cos\frac{\phi_3}{2} + \hat{\boldsymbol{e}}_3'' \sin\frac{\phi_3}{2} \\
&= \cos\frac{\phi_3}{2} + q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2)\hat{\boldsymbol{e}}_3 q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1) \sin\frac{\phi_3}{2} \\
&= q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2)\left(\cos\frac{\phi_3}{2} + \hat{\boldsymbol{e}}_3 \sin\frac{\phi_3}{2}\right) q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1) \\
&= q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_3}(\phi_3) q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1).
\end{aligned} \tag{D-10}$$

Therefore, the total combined rotation is

$$\begin{aligned}
R = R_3 R_2 R_1 &= q_{\hat{e}_3''}(\phi_3) q_{\hat{e}_2'}(\phi_2) q_{\hat{e}_1}(\phi_1) \\
&= [q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_3}(\phi_3) q_{\hat{e}_2}^{-1}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1)][q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_1}^{-1}(\phi_1)] q_{\hat{e}_1}(\phi_1) \\
&= q_{\hat{e}_1}(\phi_1) q_{\hat{e}_2}(\phi_2) q_{\hat{e}_3}(\phi_3),
\end{aligned} \tag{D-11}$$

as was to be shown.

The program in Listing D-1 is designed to test this result.

Listing D-1. order.cpp

```
// order.cpp: demonstrate that rotation about fixed axis in reverse order is correct
//            for both rotation about distinct axes and for rotation about repeated axes

#include "Rotation.h"
#include <iostream>
#include <cstdlib>
using namespace std;

int main( int argc, char* argv[] ) {

   va::Vector i( 1., 0., 0. ), j( 0., 1., 0. ), k( 0., 0., 1. ), v( 1.2, -3.4, 6.7 );
   va::Vector i1, j1, k1, i2, j2, k2, i3, j3, k3;
   va::Rotation R, R1, R2, R3;
   va::ORDER order = va::XYZ;   // select rotation sequence about distinct axes
```

```
   double ang_1 = 0.;
   double ang_2 = 0.;
   double ang_3 = 0.;
   if ( argc == 4 ) {

      ang_1 = va::rad( atof( argv[ 1 ] ) );
      ang_2 = va::rad( atof( argv[ 2 ] ) );
      ang_3 = va::rad( atof( argv[ 3 ] ) );
   }

   R = va::Rotation( ang_1, ang_2, ang_3, order );

   cout << "The constructed rotation is " << R << endl;
   cout << "The rotated vector is       " << R * v << endl;
   cout << "The following rotations should match this:" << endl;

   R1 = va::Rotation( i, ang_1 );   // rotation about x-axis
   R2 = va::Rotation( j, ang_2 );   // rotation about y-axis
   R3 = va::Rotation( k, ang_3 );   // rotation about z-axis

   R = R1 * R2 * R3;   // note the order is the reverse: first 3, then 2, then 1
   cout << "Reverse order about fixed axes:" << endl;
   cout << "Rotation:       " << R << endl;
   cout << "Rotated vector: " << R * v << endl;

   cout << endl << "Now the conventional way via transformed axes:" << endl << endl;

   // first rotation is about i
   R1 = va::Rotation( i, ang_1 );   // rotation about x-axis
   i1 = R1 * i;
   j1 = R1 * j;
   k1 = R1 * k;

   // second rotation is about j1
   R2 = va::Rotation( j1, ang_2 );   // rotation about transformed y-axis
   i2 = R2 * i1;
   j2 = R2 * j1;
   k2 = R2 * k1;

   // third rotation is about k2
   R3 = va::Rotation( k2, ang_3 );   // rotation about doubly transformed z-axis
   i3 = R3 * i2;
   j3 = R3 * j2;
   k3 = R3 * k2;

   R = R3 * R2 * R1;   // note the order is the original: first 1, then 2, then 3

   cout << "Original order about transformed axes:" << endl;
   cout << "Rotation:       " << R << endl;
   cout << "Rotated vector: " << R * v << endl;

   cout << endl << "This also works for repeated axes." << endl
                << "Using the same rotation angles, let’s do the whole thing over again." << endl << endl;
   order = va::XYX;   // select rotation sequence about repeated axes

   R = va::Rotation( ang_1, ang_2, ang_3, order );

   cout << "The constructed rotation is " << R << endl;
   cout << "The rotated vector is       " << R * v << endl;
   cout << "The following rotations should match this:" << endl;

   R1 = va::Rotation( i, ang_1 );   // rotation about x-axis
   R2 = va::Rotation( j, ang_2 );   // rotation about y-axis
   R3 = va::Rotation( i, ang_3 );   // rotation about x-axis

   R = R1 * R2 * R3;   // note the order is the reverse: first 3, then 2, then 1
   cout << "Reverse order about fixed axes:" << endl;
   cout << "Rotation:       " << R << endl;
   cout << "Rotated vector: " << R * v << endl;

   cout << endl << "Now the conventional way via transformed axes:" << endl << endl;

   // first rotation is about i
   R1 = va::Rotation( i, ang_1 );   // rotation about x-axis
   i1 = R1 * i;
   j1 = R1 * j;
   k1 = R1 * k;

   // second rotation is about j1
   R2 = va::Rotation( j1, ang_2 );   // rotation about transformed y-axis
   i2 = R2 * i1;
   j2 = R2 * j1;
   k2 = R2 * k1;

   // third rotation is about i2
```

```
    R3 = va::Rotation( i2, ang_3 );   // rotation about doubly transformed x-axis
    i3 = R3 * i2;
    j3 = R3 * j2;
    k3 = R3 * k2;

    R = R3 * R2 * R1;   // note the order is the original: first 1, then 2, then 3

    cout << "Original order about transformed axes:" << endl;
    cout << "Rotation:       " << R << endl;
    cout << "Rotated vector: " << R * v << endl;

    return EXIT_SUCCESS;
}
```

The command

```
./order 35. -15. 60.
```

will give the following results:

```
The constructed rotation is -0.533171 -0.0686517 -0.843218      73.8825
The rotated vector is       -0.530512 1.81621 7.36953
The following rotations should match this:
Reverse order about fixed axes:
Rotation:       -0.533171 -0.0686517 -0.843218  73.8825
Rotated vector: -0.530512 1.81621 7.36953

Now the conventional way via transformed axes:

Original order about transformed axes:
Rotation:       -0.533171 -0.0686517 -0.843218  73.8825
Rotated vector: -0.530512 1.81621 7.36953

This also works for repeated axes.
Using the same rotation angles, we do the whole thing over again.

The constructed rotation is -0.984429 0.171618 0.0380469        95.8951
The rotated vector is       2.78824 6.66967 2.37301
The following rotations should match this:
Reverse order about fixed axes:
Rotation:       -0.984429 0.171618 0.0380469    95.8951
Rotated vector: 2.78824 6.66967 2.37301

Now the conventional way via transformed axes:

Original order about transformed axes:
Rotation:       -0.984429 0.171618 0.0380469    95.8951
Rotated vector: 2.78824 6.66967 2.37301
```

INTENTIONALLY LEFT BLANK.

Appendix E. Factoring a Rotation into a Rotation Sequence

E-1. Distinct Principal Axis Factorization

The most common rotation sequence is probably the aerospace sequence, which consists of *yaw* about the body z-axis, *pitch* about the body y-axis, and *roll* about the body x-axis—in that order. However, there are a total of 6 such *distinct principal axis* rotation sequences and we will factor each one.[1]

Let us begin with the z-y-x (aerospace) rotation sequence, consisting of yaw about the z-axis, followed by pitch about the y-axis and ending with roll about the x-axis.

Let the given rotation be represented by the quaternion

$$p = p_0 + \hat{\mathbf{i}}p_1 + \hat{\mathbf{j}}p_2 + \hat{\mathbf{k}}p_3. \tag{E-1}$$

In the notation of Kuipers[1] (see pp. 194–196), we want to factor this as $a^3b^2c^1$, so we write

$$p = (a_0 + \hat{\mathbf{k}}a_3)(b_0 + \hat{\mathbf{j}}b_2)(c_0 + \hat{\mathbf{i}}c_1). \tag{E-2}$$

Let q represent the first 2 factors:

$$q = (a_0 + \hat{\mathbf{k}}a_3)(b_0 + \hat{\mathbf{j}}b_2) = a_0b_0 - \hat{\mathbf{i}}a_3b_2 + \hat{\mathbf{j}}a_0b_2 + \hat{\mathbf{k}}a_3b_0. \tag{E-3}$$

Then

$$\begin{aligned} q &= p(c^1)^{-1} = (p_0 + \hat{\mathbf{i}}p_1 + \hat{\mathbf{j}}p_2 + \hat{\mathbf{k}}p_3)(c_0 - \hat{\mathbf{i}}c_1) \\ &= (p_0c_0 + p_1c_1) + \hat{\mathbf{i}}(p_1c_0 - p_0c_1) + \hat{\mathbf{j}}(p_2c_0 - p_3c_1) + \hat{\mathbf{k}}(p_3c_0 + p_2c_1), \end{aligned} \tag{E-4}$$

from which we identify

$$q_0 = p_0c_0 + p_1c_1, \quad q_1 = p_1c_0 - p_0c_1, \quad q_2 = p_2c_0 - p_3c_1, \quad q_3 = p_3c_0 + p_2c_1. \tag{E-5}$$

The constraint equation for this to be a *tracking rotation sequence* follows from Eq. E-3:

$$q_0q_1 + q_2q_3 = \begin{bmatrix} q_0 & q_2 \end{bmatrix} \begin{bmatrix} q_1 \\ q_3 \end{bmatrix} = (a_0b_0)(-a_3b_2) + (a_0b_2)(a_3b_0) = 0. \tag{E-6}$$

[1]Kuipers JB. Quaternions and rotation sequences: a primer with applications to orbits, aerosp ace, and virtual reality. Princeton (NJ): Princeton University Press; 2002.

Now, from Eq. E-5,

$$\begin{bmatrix} q_0 \\ q_2 \end{bmatrix} = \begin{bmatrix} p_0c_0 + p_1c_1 \\ p_2c_0 - p_3c_1 \end{bmatrix} = \begin{bmatrix} p_0 & p_1 \\ p_2 & -p_3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \tag{E-7}$$

and

$$\begin{bmatrix} q_1 \\ q_3 \end{bmatrix} = \begin{bmatrix} p_1c_0 - p_0c_1 \\ p_3c_0 + p_2c_1 \end{bmatrix} = \begin{bmatrix} p_1 & -p_0 \\ p_3 & p_2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}, \tag{E-8}$$

so the constraint equation, Eq. E-6, may be written as

$$\begin{aligned} &\begin{bmatrix} c_0 & c_1 \end{bmatrix} \begin{bmatrix} p_0 & p_2 \\ p_1 & -p_3 \end{bmatrix} \begin{bmatrix} p_1 & -p_0 \\ p_3 & p_2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \\ &\begin{bmatrix} c_0 & c_1 \end{bmatrix} \begin{bmatrix} p_0p_1 + p_2p_3 & -p_0^2 + p_2^2 \\ p_1^2 - p_3^2 & -p_0p_1 - p_2p_3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = 0. \end{aligned} \tag{E-9}$$

Define the quantities

$$A = p_0p_1 + p_2p_3, \quad B = -p_0^2 + p_2^2, \quad D = p_1^2 - p_3^2. \tag{E-10}$$

Then the constraint equation becomes

$$\begin{bmatrix} c_0 & c_1 \end{bmatrix} \begin{bmatrix} A & B \\ D & -A \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = A(c_0^2 - c_1^2) + (B + D)c_0c_1 = 0. \tag{E-11}$$

Finally, this may be written as

$$-\frac{2A}{B+D} = \frac{2c_0c_1}{c_0^2 - c_1^2} = \frac{2\cos\dfrac{\phi_3}{2}\sin\dfrac{\phi_3}{2}}{\cos^2\dfrac{\phi_3}{2} - \sin^2\dfrac{\phi_3}{2}} = \frac{\sin\phi_3}{\cos\phi_3} = \tan\phi_3, \tag{E-12}$$

where we used

$$c_0 = \cos\frac{\phi_3}{2} \quad \text{and} \quad c_1 = \sin\frac{\phi_3}{2}. \tag{E-13}$$

Therefore, the final rotation (roll angle in this case) is

$$\phi_3 = \tan^{-1}\left(\frac{-2A}{B+D}\right), \tag{E-14}$$

and this quantity is known since A, B, and D are known from Eq. E-10. Furthermore,

since

$$a^3 = a_0 + \hat{\mathbf{k}} a_3 = \cos\frac{\phi_1}{2} + \hat{\mathbf{k}}\sin\frac{\phi_1}{2}, \tag{E-15}$$

it follows that

$$\tan\frac{\phi_1}{2} = \frac{a_3}{a_0} = \frac{a_3 b_0}{a_0 b_0} = \frac{q_3}{q_0}, \tag{E-16}$$

from Eq. E-3. Therefore, the first rotation (yaw angle in this case) is given by

$$\phi_1 = 2\tan^{-1}\left(\frac{q_3}{q_0}\right). \tag{E-17}$$

Similarly,

$$\tan\frac{\phi_2}{2} = \frac{b_2}{b_0} = \frac{a_0 b_2}{a_0 b_0} = \frac{q_2}{q_0}, \tag{E-18}$$

again using Eq. E-3, and therefore the second rotation (pitch angle in this case) is given by

$$\phi_2 = 2\tan^{-1}\left(\frac{q_2}{q_0}\right). \tag{E-19}$$

In summary, the prescription for factoring an arbitrary rotation into yaw (about the z-axis), pitch (about the y-axis), and roll (about the x-axis) (in that order) is given in Table E-1.

Table E-1. Factorization into z-y-x (aerospace) rotation sequence, consisting of yaw about the z-axis, pitch about the y-axis, and roll about the x-axis

$A = p_0p_1 + p_2p_3, \quad B = p_2^2 - p_0^2, \quad D = p_1^2 - p_3^2$
$\phi_3 = \tan^{-1}\left(\frac{-2A}{B+D}\right)$ [third rotation, roll about x-axis]
$c_0 = \cos\frac{\phi_3}{2}, \quad c_1 = \sin\frac{\phi_3}{2}$
$q_0 = p_0c_0 + p_1c_1, \quad q_1 = p_1c_0 - p_0c_1, \quad q_2 = p_2c_0 - p_3c_1, \quad q_3 = p_3c_0 + p_2c_1$
$\phi_1 = 2\tan^{-1}\left(\frac{q_3}{q_0}\right)$ [first rotation, yaw about z-axis]
$\phi_2 = 2\tan^{-1}\left(\frac{q_2}{q_0}\right)$ [second rotation, pitch about y-axis]

The calculations for the other 5 sequential orders are entirely similar and we simply summarize the results in Tables E-2 through E-6.

Table E-2. Factorization into x-y-z (FATEPEN) rotation sequence, consisting of pitch about the x-axis, yaw about the y-axis, and roll about the z-axis

$A = p_1p_2 - p_0p_3, \quad B = p_1^2 - p_3^2, \quad D = p_0^2 - p_2^2$
$\phi_3 = \tan^{-1}\left(\dfrac{-2A}{B+D}\right)$ [third rotation, roll about z-axis]
$c_0 = \cos\dfrac{\phi_3}{2}, \quad c_3 = \sin\dfrac{\phi_3}{2}$
$q_0 = p_0c_0 + p_3c_3, \quad q_1 = p_1c_0 - p_2c_3, \quad q_2 = p_2c_0 + p_1c_3, \quad q_3 = p_3c_0 - p_0c_3$
$\phi_1 = 2\tan^{-1}\left(\dfrac{q_1}{q_0}\right)$ [first rotation, pitch about x-axis]
$\phi_2 = 2\tan^{-1}\left(\dfrac{q_2}{q_0}\right)$ [second rotation, yaw about y-axis]

Table E-3. Factorization into y-x-z rotation sequence

$A = p_1p_2 + p_0p_3, \quad B = p_1^2 + p_3^2, \quad D = -p_0^2 - p_2^2$
$\phi_3 = \tan^{-1}\left(\dfrac{-2A}{B+D}\right)$ [third rotation about z-axis]
$c_0 = \cos\dfrac{\phi_3}{2}, \quad c_3 = \sin\dfrac{\phi_3}{2}$
$q_0 = p_0c_0 + p_3c_3, \quad q_1 = p_1c_0 - p_2c_3, \quad q_2 = p_2c_0 + p_1c_3, \quad q_3 = p_3c_0 - p_0c_3$
$\phi_1 = 2\tan^{-1}\left(\dfrac{q_2}{q_0}\right)$ [first rotation about y-axis]
$\phi_2 = 2\tan^{-1}\left(\dfrac{q_1}{q_0}\right)$ [second rotation about x-axis]

Table E-4. Factorization into z-x-y rotation sequence

$A = p_1p_3 - p_0p_2, \quad B = p_0^2 - p_1^2, \quad D = p_3^2 - p_2^2$
$\phi_3 = \tan^{-1}\left(\dfrac{-2A}{B+D}\right)$ [third rotation about y-axis]
$c_0 = \cos\dfrac{\phi_3}{2}, \quad c_2 = \sin\dfrac{\phi_3}{2}$
$q_0 = p_0c_0 + p_2c_2, \quad q_1 = p_1c_0 + p_3c_2, \quad q_2 = p_2c_0 - p_0c_2, \quad q_3 = p_3c_0 - p_1c_2$
$\phi_1 = 2\tan^{-1}\left(\dfrac{q_3}{q_0}\right)$ [first rotation about z-axis]
$\phi_2 = 2\tan^{-1}\left(\dfrac{q_1}{q_0}\right)$ [second rotation about x-axis]

Table E-5. Factorization into x-z-y rotation sequence

$A = p_1p_3 + p_0p_2, \quad B = -p_0^2 - p_1^2, \quad D = p_2^2 + p_3^2$
$\phi_3 = \tan^{-1}\left(\dfrac{-2A}{B+D}\right)$ [third rotation about y-axis]
$c_0 = \cos\dfrac{\phi_3}{2}, \quad c_2 = \sin\dfrac{\phi_3}{2}$
$q_0 = p_0c_0 + p_2c_2, \quad q_1 = p_1c_0 + p_3c_2, \quad q_2 = p_2c_0 - p_0c_2, \quad q_3 = p_3c_0 - p_1c_2$
$\phi_1 = 2\tan^{-1}\left(\dfrac{q_1}{q_0}\right)$ [first rotation about x-axis]
$\phi_2 = 2\tan^{-1}\left(\dfrac{q_3}{q_0}\right)$ [second rotation about z-axis]

Table E-6. Factorization into y-z-x rotation sequence

$A = p_2p_3 - p_0p_1, \quad B = p_0^2 + p_2^2, \quad D = -p_1^2 - p_3^2$
$\phi_3 = \tan^{-1}\left(\dfrac{-2A}{B+D}\right)$ [third rotation about x-axis]
$c_0 = \cos\dfrac{\phi_3}{2}, \quad c_1 = \sin\dfrac{\phi_3}{2}$
$q_0 = p_0c_0 + p_1c_1, \quad q_1 = p_1c_0 - p_0c_1, \quad q_2 = p_2c_0 - p_3c_1, \quad q_3 = p_3c_0 + p_2c_1$
$\phi_1 = 2\tan^{-1}\left(\dfrac{q_2}{q_0}\right)$ [first rotation about y-axis]
$\phi_2 = 2\tan^{-1}\left(\dfrac{q_3}{q_0}\right)$ [second rotation about z-axis]

E-2. Repeated Principal Axis Factorization

We define a *repeated principal axis* sequence as first a rotation about one of the principal body axes, then a second rotation about another body axis, and finally a third rotation about the first body axes. There are a total of 6 such rotation sequences and we will factor each one.[1]

We begin with the z-y-z rotation sequence, consisting of first about the z-axis, second about the y-axis and third about the z-axis

Let the given rotation be represented by the quaternion

$$p = p_0 + \hat{\mathbf{i}}p_1 + \hat{\mathbf{j}}p_2 + \hat{\mathbf{k}}p_3. \tag{E-20}$$

In the notation of Kuipers[1], we want to factor this as $a^3b^2c^3$, so we write

$$p = (a_0 + \hat{\mathbf{k}}a_3)(b_0 + \hat{\mathbf{j}}b_2)(c_0 + \hat{\mathbf{k}}c_3). \tag{E-21}$$

Let q represent the first 2 factors:

$$q = (a_0 + \hat{\mathbf{k}}a_3)(b_0 + \hat{\mathbf{j}}b_2) = a_0b_0 - \hat{\mathbf{i}}a_3b_2 + \hat{\mathbf{j}}a_0b_2 + \hat{\mathbf{k}}a_3b_0. \tag{E-22}$$

Then

$$\begin{aligned} q &= p(c^1)^{-1} = (p_0 + \hat{\mathbf{i}}p_1 + \hat{\mathbf{j}}p_2 + \hat{\mathbf{k}}p_3)(c_0 - \hat{\mathbf{k}}c_3) \\ &= (p_0c_0 + p_3c_3) + \hat{\mathbf{i}}(p_1c_0 - p_2c_3) + \hat{\mathbf{j}}(p_2c_0 + p_1c_3) + \hat{\mathbf{k}}(p_3c_0 - p_0c_3), \end{aligned} \tag{E-23}$$

from which we identify

$$q_0 = p_0c_0 + p_3c_3, \quad q_1 = p_1c_0 - p_2c_3, \quad q_2 = p_2c_0 + p_1c_3, \quad q_3 = p_3c_0 - p_0c_3. \tag{E-24}$$

The constraint equation for this to be a *tracking rotation sequence* follows from Eq. E-22:

$$q_0q_1 + q_2q_3 = \begin{bmatrix} q_0 & q_2 \end{bmatrix} \begin{bmatrix} q_1 \\ q_3 \end{bmatrix} = (a_0b_0)(-a_3b_2) + (a_0b_2)(a_3b_0) = 0. \tag{E-25}$$

[1] See Kuipers, pp. 200–201, for the technique, but note that there is a typo in Eq. 8.31, which leads to an error in Eq. 8.32. This has been corrected here.

Now, from Eq. E-24,

$$\begin{bmatrix} q_0 \\ q_2 \end{bmatrix} = \begin{bmatrix} p_0 c_0 + p_3 c_3 \\ p_2 c_0 + p_1 c_3 \end{bmatrix} = \begin{bmatrix} p_0 & p_3 \\ p_2 & p_1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_3 \end{bmatrix} \tag{E-26}$$

and

$$\begin{bmatrix} q_1 \\ q_3 \end{bmatrix} = \begin{bmatrix} p_1 c_0 - p_2 c_3 \\ p_3 c_0 - p_0 c_3 \end{bmatrix} = \begin{bmatrix} p_1 & -p_2 \\ p_3 & -p_0 \end{bmatrix} \begin{bmatrix} c_0 \\ c_3 \end{bmatrix}, \tag{E-27}$$

so that the constraint equation, Eq. E-25, may be written as

$$\begin{aligned} &\begin{bmatrix} c_0 & c_3 \end{bmatrix} \begin{bmatrix} p_0 & p_2 \\ p_3 & p_1 \end{bmatrix} \begin{bmatrix} p_1 & -p_2 \\ p_3 & -p_0 \end{bmatrix} \begin{bmatrix} c_0 \\ c_3 \end{bmatrix} = \\ &\begin{bmatrix} c_0 & c_3 \end{bmatrix} \begin{bmatrix} p_0 p_1 + p_2 p_3 & -p_0 p_2 - p_0 p_2 \\ p_1 p_3 + p_1 p_3 & -p_2 p_3 - p_0 p_1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_3 \end{bmatrix} = 0. \end{aligned} \tag{E-28}$$

Define the quantities

$$A = p_0 p_1 + p_2 p_3, \quad B = -2 p_0 p_2, \quad D = 2 p_1 p_3. \tag{E-29}$$

Then the constraint equation becomes

$$\begin{bmatrix} c_0 & c_3 \end{bmatrix} \begin{bmatrix} A & B \\ D & -A \end{bmatrix} \begin{bmatrix} c_0 \\ c_3 \end{bmatrix} = A(c_0^2 - c_3^2) + (B + D) c_0 c_3 = 0. \tag{E-30}$$

Finally, this may be written as

$$-\frac{2A}{B + D} = \frac{2 c_0 c_3}{c_0^2 - c_3^2} = \frac{2 \cos \dfrac{\phi_3}{2} \sin \dfrac{\phi_3}{2}}{\cos^2 \dfrac{\phi_3}{2} - \sin^2 \dfrac{\phi_3}{2}} = \frac{\sin \phi_3}{\cos \phi_3} = \tan \phi_3, \tag{E-31}$$

where we used

$$c_0 = \cos \frac{\phi_3}{2} \quad \text{and} \quad c_3 = \sin \frac{\phi_3}{2}. \tag{E-32}$$

Therefore, the final rotation is

$$\phi_3 = \tan^{-1} \left(\frac{-2A}{B + D} \right), \tag{E-33}$$

and this quantity is known since A, B, and D are known from Eq. E-29. Furthermore,

since

$$a^3 = a_0 + \hat{\mathbf{k}}a_3 = \cos\frac{\phi_1}{2} + \hat{\mathbf{k}}\sin\frac{\phi_1}{2}, \tag{E-34}$$

it follows that

$$\tan\frac{\phi_1}{2} = \frac{a_3}{a_0} = \frac{a_3 b_0}{a_0 b_0} = \frac{q_3}{q_0}, \tag{E-35}$$

where we used Eq. E-22. Therefore, the first rotation is given by

$$\phi_1 = 2\tan^{-1}\left(\frac{q_3}{q_0}\right). \tag{E-36}$$

Similarly,

$$\tan\frac{\phi_2}{2} = \frac{b_2}{b_0} = \frac{a_0 b_2}{a_0 b_0} = \frac{q_2}{q_0}, \tag{E-37}$$

again using Eq. E-22, and therefore the second rotation (pitch angle in this case) is given by

$$\phi_2 = 2\tan^{-1}\left(\frac{q_2}{q_0}\right). \tag{E-38}$$

In summary, the prescription for factoring an arbitrary rotation into an Euler sequence of first a rotation about the z-axis, followed by a rotation about the body y-axis, and finally ending with a rotation about the body z-axis, is given in Table E-7.

Table E-7. Factorization into z-y-x rotation sequence

$A = p_0p_1 + p_2p_3, \quad B = -2p_0p_2, \quad D = 2p_1p_3$
$\phi_3 = \tan^{-1}\left(\frac{-2A}{B+D}\right)$ [third rotation about z-axis]
$c_0 = \cos\frac{\phi_3}{2}, \quad c_3 = \sin\frac{\phi_3}{2}$
$q_0 = p_0c_0 + p_3c_3, \quad q_1 = p_1c_0 - p_2c_3, \quad q_2 = p_2c_0 + p_1c_3, \quad q_3 = p_3c_0 - p_0c_3$
$\phi_1 = 2\tan^{-1}\left(\frac{q_3}{q_0}\right)$ [first rotation about z-axis]
$\phi_2 = 2\tan^{-1}\left(\frac{q_2}{q_0}\right)$ [second rotation about y-axis]

The calculations for the other 5 sequential orders are entirely similar and we simply summarize the results in Tables E-8 through E-12.

Table E-8. Factorization into z-x-z rotation sequence

$A = p_0 p_2 - p_1 p_3, \quad B = 2p_0 p_1, \quad D = 2p_2 p_3$
$\phi_3 = \tan^{-1}\left(\frac{-2A}{B+D}\right)$ [third rotation about z-axis]
$c_0 = \cos\frac{\phi_3}{2}, \quad c_3 = \sin\frac{\phi_3}{2}$
$q_0 = p_0c_0 + p_3c_3, \quad q_1 = p_1c_0 - p_2c_3, \quad q_2 = p_2c_0 + p_1c_3, \quad q_3 = p_3c_0 - p_0c_3$
$\phi_1 = 2\tan^{-1}\left(\frac{q_3}{q_0}\right)$ [first rotation about z-axis]
$\phi_2 = 2\tan^{-1}\left(\frac{q_1}{q_0}\right)$ [second rotation about x-axis]

Table E-9. Factorization into y-z-y rotation sequence

$A = p_0 p_1 - p_2 p_3, \quad B = p_0 p_3 + p_1 p_2$
$\phi_3 = \tan^{-1}\left(\frac{-A}{B}\right)$ [third rotation about y-axis]
$c_0 = \cos\frac{\phi_3}{2}, \quad c_2 = \sin\frac{\phi_3}{2}$
$q_0 = p_0c_0 + p_2c_2, \quad q_1 = p_1c_0 + p_3c_2, \quad q_2 = p_2c_0 - p_0c_2, \quad q_3 = p_3c_0 - p_1c_2$
$\phi_1 = 2\tan^{-1}\left(\frac{q_2}{q_0}\right)$ [first rotation about y-axis]
$\phi_2 = 2\tan^{-1}\left(\frac{q_3}{q_0}\right)$ [second rotation about z-axis]

Table E-10. Factorization into y-x-y rotation sequence

$A = p_0 p_3 + p_1 p_2, \quad B = -2p_0 p_1, \quad D = 2p_2 p_3$
$\phi_3 = \tan^{-1}\left(\frac{-2A}{B+D}\right)$ [third rotation about y-axis]
$c_0 = \cos\frac{\phi_3}{2}, \quad c_2 = \sin\frac{\phi_3}{2}$
$q_0 = p_0c_0 + p_2c_2, \quad q_1 = p_1c_0 + p_3c_2, \quad q_2 = p_2c_0 - p_0c_2, \quad q_3 = p_3c_0 - p_1c_2$
$\phi_1 = 2\tan^{-1}\left(\frac{q_2}{q_0}\right)$ [first rotation about y-axis]
$\phi_2 = 2\tan^{-1}\left(\frac{q_1}{q_0}\right)$ [second rotation about x-axis]

Table E-11. Factorization into x-y-x rotation sequence

$A = p_0p_3 - p_1p_2, \quad B = p_0p_2 + p_1p_3$
$\phi_3 = \tan^{-1}\left(\frac{-A}{B}\right)$ [third rotation about x-axis]
$c_0 = \cos\frac{\phi_3}{2}, \quad c_1 = \sin\frac{\phi_3}{2}$
$q_0 = p_0c_0 + p_1c_1, \quad q_1 = p_1c_0 - p_0c_1, \quad q_2 = p_2c_0 - p_3c_1, \quad q_3 = p_3c_0 + p_2c_1$
$\phi_1 = 2\tan^{-1}\left(\frac{q_1}{q_0}\right)$ [first rotation about x-axis]
$\phi_2 = 2\tan^{-1}\left(\frac{q_2}{q_0}\right)$ [second rotation about y-axis]

Table E-12. Factorization into x-z-x rotation sequence

$A = p_0p_2 + p_1p_3, \quad B = -p_0p_3 + p_1p_2$
$\phi_3 = \tan^{-1}\left(\frac{-A}{B}\right)$ [third rotation about x-axis]
$c_0 = \cos\frac{\phi_3}{2}, \quad c_1 = \sin\frac{\phi_3}{2}$
$q_0 = p_0c_0 + p_1c_1, \quad q_1 = p_1c_0 - p_0c_1, \quad q_2 = p_2c_0 - p_3c_1, \quad q_3 = p_3c_0 + p_2c_1$
$\phi_1 = 2\tan^{-1}\left(\frac{q_1}{q_0}\right)$ [first rotation about x-axis]
$\phi_2 = 2\tan^{-1}\left(\frac{q_3}{q_0}\right)$ [second rotation about z-axis]

The program in Listing E-1 is designed to test these formulas.

Listing E-1. factor.cpp

```
// factor.cpp: test program for the rotation factorization

#include "Rotation.h"
#include <iostream>
#include <cstdlib>

int main( int argc, char* argv[] ) {

   va::Rotation R;
   rng::Random rng;
   va::ORDER order = va::ORDER( rng.uniformDiscrete( 0, 11 ) );
   double ang_1, ang_2, ang_3;
   if ( argc == 4 ) {

      ang_1 = va::rad( atof( argv[ 1 ] ) );
      ang_2 = va::rad( atof( argv[ 2 ] ) );
      ang_3 = va::rad( atof( argv[ 3 ] ) );
      R = va::Rotation( ang_1, ang_2, ang_3, order );
      std::cout << "order = " << order << std::endl;
```

```
    }
    else if ( argc == 1 ) {
        R = va::Rotation( rng );
    }
    else {
        std::cerr << argv[ 0 ] << " usage: Enter angles 1, 2, and 3 (deg) on commandline" << std::endl;
        exit( EXIT_FAILURE );
    }
    std::cout << "The rotation is " << R << std::endl << std::endl;;   // output the rotation
    std::cout << "The following rotations should match this" << std::endl;

    for ( int order = 0; order < 12; order++ ) {

        va::sequence s = va::factor( R, va::ORDER( order ) );   // factor the rotation

        std::cout << "1st rotation = " << va::deg( s.first ) << "\torder = " << order << std::endl;   // output the factorization
        std::cout << "2nd rotation = " << va::deg( s.second ) << std::endl;
        std::cout << "3rd rotation = " << va::deg( s.third ) << std::endl;

        R = va::Rotation( s, va::ORDER( order ) );   // generate the rotation with this sequence
        std::cout << R << std::endl;                 // output the rotation so that it can be compared
    }
    return EXIT_SUCCESS;
}
```

The command

```
./factor
```

will generate a random rotation, so each run will be different. But the factored rotation must match the randomly generated rotation, as shown here:

```
The rotation is 0.18777 0.958993 -0.212307      86.6526

The following rotations should match this
1st rotation = -24.8355 order = 0
2nd rotation = 84.2077
3rd rotation = -2.42093
0.18777 0.958993 -0.212307      86.6526
1st rotation = 75.1077  order = 1
2nd rotation = 66.8997
3rd rotation = -76.5002
0.18777 0.958993 -0.212307      86.6526
1st rotation = 83.7441  order = 2
2nd rotation = 22.2818
3rd rotation = -2.62561
0.18777 0.958993 -0.212307      86.6526
1st rotation = -22.4269 order = 3
2nd rotation = -0.244254
3rd rotation = 84.2128
0.18777 0.958993 -0.212307      86.6526
1st rotation = -0.264239        order = 4
2nd rotation = -22.4267
3rd rotation = 84.3136
0.18777 0.958993 -0.212307      86.6526
1st rotation = 84.7402  order = 5
2nd rotation = -2.42944
3rd rotation = 22.303
0.18777 0.958993 -0.212307      86.6526
1st rotation = -22.4022 order = 6
2nd rotation = 84.2129
3rd rotation = -0.245506
0.18777 0.958993 -0.212307      86.6526
1st rotation = -112.402 order = 7
2nd rotation = -84.2129
3rd rotation = 89.7545
0.18777 0.958993 -0.212307      86.6526
1st rotation = 0.640217 order = 8
2nd rotation = -22.4282
3rd rotation = 83.621
0.18777 0.958993 -0.212307      86.6526
1st rotation = 90.6402  order = 9
2nd rotation = 22.4282
3rd rotation = -6.37896
0.18777 0.958993 -0.212307      86.6526
1st rotation = -2.4397  order = 10
2nd rotation = 84.745
```

```
3rd rotation = 22.5265
0.18777 0.958993 -0.212307      86.6526
1st rotation = 87.5603  order = 11
2nd rotation = -84.745
3rd rotation = -67.4735
0.18777 0.958993 -0.212307      86.6526
```

We can also input an explicit rotation:

```
./factor -13. 67. -23.
```

This gives the following results:

```
order = 1
The rotation is -0.33597 0.863186 -0.376875     73.8475

The following rotations should match this
1st rotation = -57.8081 order = 0
2nd rotation = 47.5373
3rd rotation = -55.6718
-0.33597 0.863186 -0.376875     73.8475
1st rotation = -13      order = 1
2nd rotation = 67
3rd rotation = -23
-0.33597 0.863186 -0.376875     73.8475
1st rotation = 67.5302  order = 2
2nd rotation = -5.04254
3rd rotation = -34.9977
-0.33597 0.863186 -0.376875     73.8475
1st rotation = -10.5973 order = 3
2nd rotation = -33.8845
3rd rotation = 62.7029
-0.33597 0.863186 -0.376875     73.8475
1st rotation = -34.3421 order = 4
2nd rotation = -8.78174
3rd rotation = 68.6579
-0.33597 0.863186 -0.376875     73.8475
1st rotation = 64.0087  order = 5
2nd rotation = -34.8426
3rd rotation = -6.14787
-0.33597 0.863186 -0.376875     73.8475
1st rotation = 5.45441  order = 6
2nd rotation = 67.6219
3rd rotation = -37.0797
-0.33597 0.863186 -0.376875     73.8475
1st rotation = -84.5456 order = 7
2nd rotation = -67.6219
3rd rotation = 52.9203
-0.33597 0.863186 -0.376875     73.8475
1st rotation = 74.6856  order = 8
2nd rotation = -35.3132
3rd rotation = -8.7461
-0.33597 0.863186 -0.376875     73.8475
1st rotation = -15.3144 order = 9
2nd rotation = -35.3132
3rd rotation = 81.2539
-0.33597 0.863186 -0.376875     73.8475
1st rotation = -37.756  order = 10
2nd rotation = 68.9201
3rd rotation = 9.4171
-0.33597 0.863186 -0.376875     73.8475
1st rotation = 52.244   order = 11
2nd rotation = -68.9201
3rd rotation = -80.5829
-0.33597 0.863186 -0.376875     73.8475
```

The order variable is arbitrary so it is randomized in the program. It is output so that we can check that it found the same rotation sequence for that particular order (in this case, order = 1).

INTENTIONALLY LEFT BLANK.

Appendix F. Conversion between Quaternion and Rotation Matrix

F-1. Quaternion to Rotation Matrix

For rotations about a principal axis, the correspondence is as follows:

- Rotation about the x-axis: $\cos\frac{\theta}{2} + \hat{\mathbf{\imath}}\sin\frac{\theta}{2} \quad \Leftrightarrow \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$ (F-1)
- Rotation about the y-axis: $\cos\frac{\theta}{2} + \hat{\mathbf{\jmath}}\sin\frac{\theta}{2} \quad \Leftrightarrow \quad \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$ (F-2)
- Rotation about the z-axis: $\cos\frac{\theta}{2} + \hat{\mathbf{k}}\sin\frac{\theta}{2} \quad \Leftrightarrow \quad \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$ (F-3)

In general, if the quaternion is given by

$$q = q_0 + q_1\hat{\mathbf{\imath}} + q_2\hat{\mathbf{\jmath}} + q_3\hat{\mathbf{k}}, \tag{F-4}$$

then the corresponding rotation matrix is

$$A = \begin{bmatrix} 2q_0^2 - 1 + 2q_1^2 & 2q_1q_2 - 2q_0q_3 & 2q_1q_3 + 2q_0q_2 \\ 2q_1q_2 + 2q_0q_3 & 2q_0^2 - 1 + 2q_2^2 & 2q_2q_3 - 2q_0q_1 \\ 2q_1q_3 - 2q_0q_2 & 2q_2q_3 + 2q_0q_1 & 2q_0^2 - 1 + 2q_3^2 \end{bmatrix}. \tag{F-5}$$

F-2. Rotation Matrix to Quaternion

Let the rotation be given by the matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}. \tag{F-6}$$

Then a vector along the axis of rotation is

$$\boldsymbol{v} = (a_{32} - a_{23})\hat{\mathbf{\imath}} + (a_{13} - a_{31})\hat{\mathbf{\jmath}} + (a_{21} - a_{12})\hat{\mathbf{k}}. \tag{F-7}$$

If this vector turns out to be zero, then A is the identity matrix. Otherwise, a unit vector along the axis of rotation is

$$\hat{\boldsymbol{u}} = \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|} = \frac{(a_{32} - a_{23})\hat{\mathbf{i}} + (a_{13} - a_{31})\hat{\mathbf{j}} + (a_{21} - a_{12})\hat{\mathbf{k}}}{\sqrt{(a_{32} - a_{23})^2 + (a_{13} - a_{31})^2 + (a_{21} - a_{12})^2}}. \tag{F-8}$$

The rotation angle is

$$\theta = \cos^{-1}\left(\frac{a_{11} + a_{22} + a_{33} - 1}{2}\right). \tag{F-9}$$

And the corresponding quaternion is

$$q = \cos\frac{\theta}{2} + \hat{\boldsymbol{u}}\sin\frac{\theta}{2}. \tag{F-10}$$

F-3. Conversion between Rotation, Rotation Matrix, and Quaternion

The program in Listing F-1 will convert between the 3 different representations.

Listing F-1. convert.cpp

```cpp
// convert.cpp: convert between Rotation, rotation matix, and quaternion

#include "Rotation.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>
using namespace va;

int main( int argc, char* argv[] ) {

   // specify a Rotation from an axis vector and rotation angle
   Vector u = normalize( Vector( 1., 1., 1. ) );
   double th = rad( 120. );

   if ( argc == 5 ) {

      u = normalize( Vector( atof( argv[1] ), atof( argv[2] ), atof( argv[3] ) ) );
      th = rad( atof( argv[4] ) );
   }

   std::cout << std::setprecision(6) << std::fixed << std::showpos;

   Rotation R( u, th );
   matrix A;
   quaternion q;
   std::cout << "Given the Rotation:" << std::endl;
   std::cout << R << std::endl << std::endl;

   // convert Rotation to a rotation matrix
   std::cout << "convert Rotation to rotation matrix:" << std::endl;
   A = to_matrix( R );
   std::cout << A << std::endl << std::endl;

   // convert a Rotation to a quaternion
   std::cout << "convert Rotation to quaternion:" << std::endl;
   q = to_quaternion( R );
   std::cout << q << std::endl << std::endl;

   std::cout << "Given the rotation matrix:" << std::endl;
   std::cout << A << std::endl << std::endl;

   //convert a rotation matrix to a Rotation
```

```
    std::cout << "convert rotation matrix to Rotation:" << std::endl;
    R = Rotation( A );
    std::cout << R << std::endl << std::endl;

    //convert a rotation matrix to a quaternion
    std::cout << "convert rotation matrix to quaternion:" << std::endl;
    q = to_quaternion( Rotation( A ) );
    std::cout << q << std::endl << std::endl;;

    std::cout << "Given the quaternion:" << std::endl;
    std::cout << q << std::endl << std::endl;

    // convert a quaternion to a Rotation
    std::cout << "convert quaternion to Rotation:" << std::endl;
    R = Rotation( q );
    std::cout << R << std::endl << std::endl;

    // convert a quaternion to a rotation matrix
    std::cout << "convert quaternion to rotation matrix:" << std::endl;
    A = to_matrix( Rotation( q ) );
    std::cout << A << std::endl;

    return EXIT_SUCCESS;
}
```

Compiling this program with

```
g++ -O2 -Wall -o convert convert.cpp -lm
```

and then running it via the command

```
./convert 2.35 6.17 -4.6 35.6
```

produces the following output:

```
Given the Rotation:
+0.292041 +0.766762 -0.571654   +35.600000

convert Rotation to rotation matrix:
+0.829041       +0.374624       +0.415148
-0.290921       +0.922983       -0.251926
-0.477552       +0.088081       +0.874177

convert Rotation to quaternion:
+0.952129       +0.089275 +0.234396 -0.174752

Given the rotation matrix:
+0.829041       +0.374624       +0.415148
-0.290921       +0.922983       -0.251926
-0.477552       +0.088081       +0.874177

convert rotation matrix to Rotation:
+0.292041 +0.766762 -0.571654   +35.600000

convert rotation matrix to quaternion:
+0.952129       +0.089275 +0.234396 -0.174752

Given the quaternion:
+0.952129       +0.089275 +0.234396 -0.174752

convert quaternion to Rotation:
+0.292041 +0.766762 -0.571654   +35.600000

convert quaternion to rotation matrix:
+0.829041       +0.374624       +0.415148
-0.290921       +0.922983       -0.251926
-0.477552       +0.088081       +0.874177
```

Appendix G. Slerp (Spherical Linear Interpolation)

This is a derivation of the spherical linear interpolation (Slerp) formula that was introduced by Shomake.[1] As depicted in Fig. G-1, the rotation that takes the unit vector $\hat{\boldsymbol{u}}_1$ to the unit vector $\hat{\boldsymbol{u}}_2$ is the unit quaternion

$$q \equiv \cos\frac{\theta}{2} + \hat{\boldsymbol{n}}\sin\frac{\theta}{2}, \tag{G-1}$$

where θ is the angle between $\hat{\boldsymbol{u}}_1$ and $\hat{\boldsymbol{u}}_2$, and

$$\hat{\boldsymbol{n}} \equiv \frac{\hat{\boldsymbol{u}}_1 \times \hat{\boldsymbol{u}}_2}{\|\hat{\boldsymbol{u}}_1 \times \hat{\boldsymbol{u}}_2\|} \tag{G-2}$$

is the unit vector along the axis of rotation. This means that

$$\hat{\boldsymbol{u}}_2 = q\hat{\boldsymbol{u}}_1 q^{-1}. \tag{G-3}$$

Now let us parametrize the angle as $t\theta$, where $0 \le t \le 1$, and let

$$q(t) \equiv \cos\frac{t\theta}{2} + \hat{\boldsymbol{n}}\sin\frac{t\theta}{2}. \tag{G-4}$$

Then an intermediate unit vector $\hat{\boldsymbol{u}}(t)$ that runs along the arc on the unit circle from $\hat{\boldsymbol{u}}_1$ to $\hat{\boldsymbol{u}}_2$ is given by

$$\hat{\boldsymbol{u}}(t) = q(t)\hat{\boldsymbol{u}}_1 q(t)^{-1} = \left(\cos\frac{t\theta}{2} + \hat{\boldsymbol{n}}\sin\frac{t\theta}{2}\right)\hat{\boldsymbol{u}}_1\left(\cos\frac{t\theta}{2} - \hat{\boldsymbol{n}}\sin\frac{t\theta}{2}\right). \tag{G-5}$$

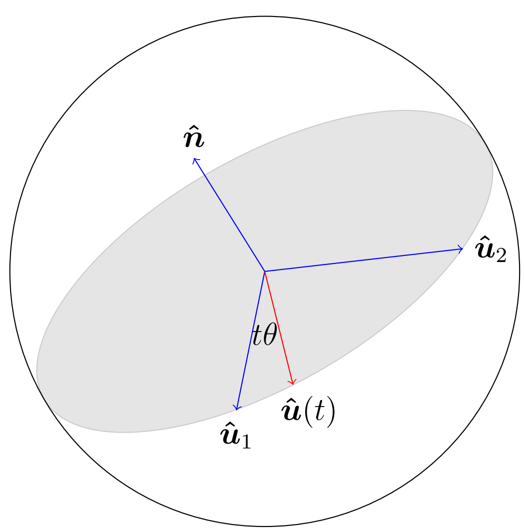

Fig. G-1. Spherical linear interpolation over the unit sphere

[1]Shoemake K. Animating rotation with quaternion curves. SIGGRAPH '85; 1985;245–254.

Treating $\hat{\boldsymbol{u}}_1$ as the pure quaternion $(0, \hat{\boldsymbol{u}}_1)$ and using the fact that $\hat{\boldsymbol{u}}_1 \cdot \hat{\boldsymbol{n}} = 0$, $\hat{\boldsymbol{u}}_2 \cdot \hat{\boldsymbol{n}} = 0$, $\hat{\boldsymbol{n}} \cdot (\hat{\boldsymbol{n}} \times \hat{\boldsymbol{u}}_1) = 0$, and $\|\hat{\boldsymbol{u}}_1 \times \hat{\boldsymbol{u}}_2\| = \sin\theta$, we can carry out the quaternion multiplication to get

$$\begin{aligned}
\hat{\boldsymbol{u}}(t) &= \left(\cos\frac{t\theta}{2} + \hat{\boldsymbol{n}}\sin\frac{t\theta}{2}\right)\hat{\boldsymbol{u}}_1\left(\cos\frac{t\theta}{2} - \hat{\boldsymbol{n}}\sin\frac{t\theta}{2}\right) \\
&= \left(\cos\frac{t\theta}{2}\hat{\boldsymbol{u}}_1 + \hat{\boldsymbol{n}} \times \hat{\boldsymbol{u}}_1 \sin\frac{t\theta}{2}\right)\left(\cos\frac{t\theta}{2} - \hat{\boldsymbol{n}}\sin\frac{t\theta}{2}\right) \\
&= \cos^2\frac{t\theta}{2}\hat{\boldsymbol{u}}_1 + \cos\frac{t\theta}{2}\sin\frac{t\theta}{2}\hat{\boldsymbol{n}} \times \hat{\boldsymbol{u}}_1 + \cos\frac{t\theta}{2}\sin\frac{t\theta}{2}\hat{\boldsymbol{n}} \times \hat{\boldsymbol{u}}_1 - \sin^2\frac{t\theta}{2}(\hat{\boldsymbol{n}} \times \hat{\boldsymbol{u}}_1) \times \hat{\boldsymbol{n}} \\
&= \cos^2\frac{t\theta}{2}\hat{\boldsymbol{u}}_1 + 2\cos\frac{t\theta}{2}\sin\frac{t\theta}{2}\hat{\boldsymbol{n}} \times \hat{\boldsymbol{u}}_1 + \sin^2\frac{t\theta}{2}\hat{\boldsymbol{n}} \times (\hat{\boldsymbol{n}} \times \hat{\boldsymbol{u}}_1).
\end{aligned} \tag{G-6}$$

Now

$$\begin{aligned}
\hat{\boldsymbol{n}} \times \hat{\boldsymbol{u}}_1 = -\hat{\boldsymbol{u}}_1 \times \hat{\boldsymbol{n}} &= -\frac{\hat{\boldsymbol{u}}_1 \times (\hat{\boldsymbol{u}}_1 \times \hat{\boldsymbol{u}}_2)}{\sin\theta} \\
&= -\frac{\hat{\boldsymbol{u}}_1(\hat{\boldsymbol{u}}_1 \cdot \hat{\boldsymbol{u}}_2) - \hat{\boldsymbol{u}}_2(\hat{\boldsymbol{u}}_1 \cdot \hat{\boldsymbol{u}}_1)}{\sin\theta} \\
&= \frac{\hat{\boldsymbol{u}}_2 - \cos\theta\,\hat{\boldsymbol{u}}_1}{\sin\theta}
\end{aligned} \tag{G-7}$$

and

$$\hat{\boldsymbol{n}} \times (\hat{\boldsymbol{n}} \times \hat{\boldsymbol{u}}_1) = \hat{\boldsymbol{n}}(\hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{u}}_1) - \hat{\boldsymbol{u}}_1(\hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{n}}) = -\hat{\boldsymbol{u}}_1 \tag{G-8}$$

so that

$$\begin{aligned}
\hat{\boldsymbol{u}}(t) &= \left(\cos^2\frac{t\theta}{2} - \sin^2\frac{t\theta}{2}\right)\hat{\boldsymbol{u}}_1 + 2\cos\frac{t\theta}{2}\sin\frac{t\theta}{2}\left(\frac{\hat{\boldsymbol{u}}_2 - \cos\theta\,\hat{\boldsymbol{u}}_1}{\sin\theta}\right) \\
&= \cos t\theta\,\hat{\boldsymbol{u}}_1 + \sin t\theta\left(\frac{\hat{\boldsymbol{u}}_2 - \cos\theta\,\hat{\boldsymbol{u}}_1}{\sin\theta}\right) \\
&= \frac{\cos t\theta\sin\theta\,\hat{\boldsymbol{u}}_1 + \sin t\theta\,\hat{\boldsymbol{u}}_2 - \sin t\theta\cos\theta\,\hat{\boldsymbol{u}}_1}{\sin\theta} \\
&= \frac{\sin(\theta - t\theta)\,\hat{\boldsymbol{u}}_1 + \sin t\theta\,\hat{\boldsymbol{u}}_2}{\sin\theta} \\
&= \frac{\sin(1-t)\theta}{\sin\theta}\hat{\boldsymbol{u}}_1 + \frac{\sin t\theta}{\sin\theta}\hat{\boldsymbol{u}}_2.
\end{aligned} \tag{G-9}$$

G-1. Slerp Formula

Thus, the spherical linear interpolation of the unit vector on the arc of the unit sphere from $\hat{\boldsymbol{u}}_1$ to $\hat{\boldsymbol{u}}_2$ is given by

$$\boxed{\hat{\boldsymbol{u}}(t) = \frac{\sin(1-t)\theta}{\sin\theta}\hat{\boldsymbol{u}}_1 + \frac{\sin t\theta}{\sin\theta}\hat{\boldsymbol{u}}_2}, \tag{G-10}$$

where $0 \le t \le 1$. This gives us the C++ implementation in Listing G-1.

Listing G-1. slerp.cpp

```
// slerp.cpp: original slerp formula

#include "Vector.h"
#include <iostream>
#include <cstdlib>
using namespace va;

int main( int argc, char* argv[] ) {

   Vector i( 1., 0., 0. ), j( 0., 1., 0. ), k( 0., 0., 1. );

   Vector u1 = i;   // default initial vector
   Vector u2 = j;   // default final vector
   if ( argc > 1 ) {   // or specify initial and final vectors on the command line

      u1 = Vector( atof( argv[1] ), atof( argv[2] ), atof( argv[3] ) );
      u2 = Vector( atof( argv[4] ), atof( argv[5] ), atof( argv[6] ) );
   }

   const int    N     = 1000;
   const double THETA = acos( u1 * u2 );
   const double A     = 1. / sin( THETA );
   Vector u;

   double t;
   for ( int n = 0; n < N; n++ ) {

      t = double( n ) / double( N-1 );
      u = A * ( sin( ( 1. - t ) * THETA ) * u1 + sin( t * THETA ) * u2 ) ;
      std::cout << u << std::endl;
   }

   return EXIT_SUCCESS;
}
```

G-2. Fast Incremental Slerp

The straightforward application of Eq. G-10, as we incrementally vary t from 0 to 1, involves the computationally expensive evaluation of trigonometric functions in an inner loop. We show here how this can be avoided.[1]

Starting with Eq. G-10, using the double angle formula, and

$$\hat{\boldsymbol{u}}_1 \cdot \hat{\boldsymbol{u}}_2 = \cos\theta, \tag{G-11}$$

[1] Barrera T, Hast A, Bengtsson E. Incremental spherical linear interpolation. SIGRAD 2004. 2005;7–10.

we have[1]

$$\begin{aligned}
\hat{\boldsymbol{u}}(t) &= \frac{[\sin\theta\cos(t\theta) - \cos\theta\sin(t\theta)]\,\hat{\boldsymbol{u}}_1 + \sin t\theta\,\hat{\boldsymbol{u}}_2}{\sin\theta} \\
&= \cos(t\theta)\,\hat{\boldsymbol{u}}_1 + \frac{\sin(t\theta)}{\sin\theta}[\hat{\boldsymbol{u}}_2 - \cos\theta\hat{\boldsymbol{u}}_1] \\
&= \cos(t\theta)\,\hat{\boldsymbol{u}}_1 + \sin(t\theta)\left[\frac{\hat{\boldsymbol{u}}_2 - \cos\theta\hat{\boldsymbol{u}}_1}{\sqrt{1-\cos^2\theta}}\right] \\
&= \cos(t\theta)\,\hat{\boldsymbol{u}}_1 + \sin(t\theta)\left[\frac{\hat{\boldsymbol{u}}_2 - (\hat{\boldsymbol{u}}_1\cdot\hat{\boldsymbol{u}}_2)\hat{\boldsymbol{u}}_1}{\sqrt{1-(\hat{\boldsymbol{u}}_1\cdot\hat{\boldsymbol{u}}_2)^2}}\right].
\end{aligned} \tag{G-12}$$

Now consider the term in square brackets. The numerator is $\hat{\boldsymbol{u}}_2$ minus the projection of $\hat{\boldsymbol{u}}_2$ onto $\hat{\boldsymbol{u}}_1$, and thus is orthogonal to $\hat{\boldsymbol{u}}_1$. Also, the denominator is the norm of the numerator, since

$$\begin{aligned}
[\hat{\boldsymbol{u}}_2 - (\hat{\boldsymbol{u}}_1\cdot\hat{\boldsymbol{u}}_2)\hat{\boldsymbol{u}}_1]\cdot[\hat{\boldsymbol{u}}_2 - (\hat{\boldsymbol{u}}_1\cdot\hat{\boldsymbol{u}}_2)\hat{\boldsymbol{u}}_1] &= 1 - (\hat{\boldsymbol{u}}_1\cdot\hat{\boldsymbol{u}}_2)^2 - (\hat{\boldsymbol{u}}_1\cdot\hat{\boldsymbol{u}}_2)^2 + (\hat{\boldsymbol{u}}_1\cdot\hat{\boldsymbol{u}}_2)^2 \\
&= 1 - (\hat{\boldsymbol{u}}_1\cdot\hat{\boldsymbol{u}}_2)^2.
\end{aligned} \tag{G-13}$$

Thus, the term in square brackets is a fixed unit vector that is tangent to $\hat{\boldsymbol{u}}_1$, which we label $\hat{\boldsymbol{u}}_0$:

$$\hat{\boldsymbol{u}}_0 \equiv \frac{\hat{\boldsymbol{u}}_2 - (\hat{\boldsymbol{u}}_1\cdot\hat{\boldsymbol{u}}_2)\hat{\boldsymbol{u}}_1}{\sqrt{1-(\hat{\boldsymbol{u}}_1\cdot\hat{\boldsymbol{u}}_2)^2}}. \tag{G-14}$$

Therefore, Eq. G-12 can be written as

$$\hat{\boldsymbol{u}}(t) = \cos(t\theta)\,\hat{\boldsymbol{u}}_1 + \sin(t\theta)\,\hat{\boldsymbol{u}}_0. \tag{G-15}$$

We want to evaluate $\hat{\boldsymbol{u}}$ incrementally, so let us discretize this equation by setting $\delta\theta = \theta/(N-1)$ and let $x = \delta\theta$. Then Eq. G-15 becomes

$$\boxed{\hat{\boldsymbol{u}}[n] = \cos(nx)\,\hat{\boldsymbol{u}}_1 + \sin(nx)\,\hat{\boldsymbol{u}}_0} \tag{G-16}$$

for $n = 0, 1, 2, \ldots, N-1$.

[1]Hast A, Barrera T, Bengtsson E. Shading by spherical linear interpolation using De Moivre's formula. WSCG'03. 2003;Short Paper;57–60.

Now we make use of the trigonometric identities

$$\begin{aligned}\cos(n+1)x + \cos(n-1)x &= 2\cos nx \cos x,\\ \sin(n+1)x + \sin(n-1)x &= 2\sin nx \cos x.\end{aligned} \tag{G-17}$$

Or, changing $n \to n-1$ and rearranging,

$$\begin{aligned}\cos nx &= 2\cos x \cos(n-1)x - \cos(n-2)x,\\ \sin nx &= 2\cos x \sin(n-1)x - \sin(n-2)x.\end{aligned} \tag{G-18}$$

Substituting these into Eq. G-16 results in a simple recurrence relation:

$$\begin{aligned}\hat{\boldsymbol{u}}[n] &= [2\cos x \cos(n-1)x - \cos(n-2)x]\,\hat{\boldsymbol{u}}_1 +\\ &\quad [2\cos x \sin(n-1)x - \sin(n-2)x]\,\hat{\boldsymbol{u}}_0\\ &= 2\cos x[\cos(n-1)x\,\hat{\boldsymbol{u}}_1 + \sin(n-1)x\,\hat{\boldsymbol{u}}_0] -\\ &\quad [\cos(n-2)x\,\hat{\boldsymbol{u}}_1 + \sin(n-2)x\,\hat{\boldsymbol{u}}_0]\\ &= 2\cos x\,\hat{\boldsymbol{u}}[n-1] - \hat{\boldsymbol{u}}[n-2].\end{aligned} \tag{G-19}$$

It is also easy to evaluate the first 2 values directly from Eq. G-16:

$$\hat{\boldsymbol{u}}[0] = \hat{\boldsymbol{u}}_1 \quad \text{and} \tag{G-20}$$

$$\hat{\boldsymbol{u}}[1] = \cos x\,\hat{\boldsymbol{u}}_1 + \sin x\,\hat{\boldsymbol{u}}_0. \tag{G-21}$$

Putting this all together gives us the C++ implementation in Listing G-2.

Listing G-2. fast_slerp.cpp

```cpp
// fast_slerp.cpp: fast incremental slerp evaluates trig functions only once
// R. Saucier, June 2016

#include "Vector.h"
#include <iostream>
#include <cstdlib>
using namespace va;

int main( int argc, char* argv[] ) {

   Vector i( 1., 0., 0. ), j( 0., 1., 0. ), k( 0., 0., 1. );

   Vector u1 = i;   // default initial vector
   Vector u2 = j;   // default final vector
   if ( argc > 1 ) {  // or specify initial and final vectors on the command line

      u1 = Vector( atof( argv[1] ), atof( argv[2] ), atof( argv[3] ) );
      u2 = Vector( atof( argv[4] ), atof( argv[5] ), atof( argv[6] ) );
   }

   const int    N     = 1000;
   const double U12   = u1 * u2;
   const double THETA = acos( U12 );
   const double TH    = THETA / double( N-1 );
   const double C     = 2. * cos( TH );
```

```
   const Vector U0    = ( u2 - U12 * u1 ) / sqrt( ( 1. - U12 ) * ( 1. +  U12 ) );

   Vector u_2, u_1, u;

   for ( int n = 0; n < N; n++ ) {

           if ( n == 0 ) u = u_2 = u1;                                // n = 0 will become u at n - 2;
      else if ( n == 1 ) u = u_1 = cos( TH ) * u1 + sin( TH ) * U0;   // n = 1 will become u at n - 1
      else {
         u   = C * u_1 - u_2;
         u_2 = u_1;
         u_1 = u;
      }
      std::cout << u << std::endl;
   }

   return EXIT_SUCCESS;
}
```

Removing the trigonometric functions from the inner loop results in a speedup of about 12 times over the original slerp formula in Eqs. G-10 or G-16.

INTENTIONALLY LEFT BLANK.

Appendix H. Exact Solution to the Absolute Orientation Problem

This solution follows the approach of Micheals and Boult.[1] Given 2 sets of 3 linearly independent vectors, $\{\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3\}$ and $\{\boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_3\}$, where the 2 sets of vectors are not necessarily *unit* vectors but are related by a pure rotation, the absolute orientation problem is to find this rotation. Since rotations can be represented by unit quaternions, there must be a unit quaternion q such that

$$\boldsymbol{b}_i = q\boldsymbol{a}_i q^{-1} \quad \text{for} \quad i = 1, 2, 3 \tag{H-1}$$

where $q = q_0 + q_1\hat{\mathbf{i}} + q_2\hat{\mathbf{j}} + q_3\hat{\mathbf{k}}$ and $q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$. Using $\hat{\mathbf{i}}^2 = \hat{\mathbf{j}}^2 = \hat{\mathbf{k}}^2 = \hat{\mathbf{i}}\hat{\mathbf{j}}\hat{\mathbf{k}} = -1$, $q^{-1} = q_0 - q_1\hat{\mathbf{i}} - q_2\hat{\mathbf{j}} - q_3\hat{\mathbf{k}}$, and expanding, we get

$$b_x = a_x q_0^2 + 2a_z q_0 q_2 - 2a_y q_0 q_3 + a_x q_1^2 + 2a_y q_1 q_2 + 2a_z q_1 q_3 - a_x q_2^2 - a_x q_3^2 \tag{H-2}$$

$$b_y = a_y q_0^2 - 2a_z q_0 q_1 + 2a_x q_0 q_3 - a_y q_1^2 + 2a_x q_1 q_2 + a_y q_2^2 + 2a_z q_2 q_3 - a_y q_3^2 \tag{H-3}$$

$$b_z = a_z q_0^2 + 2a_y q_0 q_1 - 2a_x q_0 q_2 - a_z q_1^2 + 2a_x q_1 q_3 - a_z q_2^2 + 2a_y q_2 q_3 + a_z q_3^2 \tag{H-4}$$

for each of the 3 vectors. Imposing the normalization condition then gives us 10 equations in 10 unknowns:

$$\begin{bmatrix} a_{1x} & 0 & 2a_{1z} & -2a_{1y} & a_{1x} & 2a_{1y} & 2a_{1z} & -a_{1x} & 0 & -a_{1x} \\ a_{1y} & -2a_{1z} & 0 & 2a_{1x} & -a_{1y} & 2a_{1x} & 0 & a_{1y} & 2a_{1z} & -a_{1y} \\ a_{1z} & 2a_{1y} & -2a_{1x} & 0 & -a_{1z} & 0 & 2a_{1x} & -a_{1z} & 2a_{1y} & a_{1z} \\ a_{2x} & 0 & 2a_{2z} & -2a_{2y} & a_{2x} & 2a_{2y} & 2a_{2z} & -a_{2x} & 0 & -a_{2x} \\ a_{2y} & -2a_{2z} & 0 & 2a_{2x} & -a_{2y} & 2a_{2x} & 0 & a_{2y} & 2a_{2z} & -a_{2y} \\ a_{2z} & 2a_{2y} & -2a_{2x} & 0 & -a_{2z} & 0 & 2a_{2x} & -a_{2z} & 2a_{2y} & a_{2z} \\ a_{3x} & 0 & 2a_{3z} & -2a_{3y} & a_{3x} & 2a_{3y} & 2a_{3z} & -a_{3x} & 0 & -a_{3x} \\ a_{3y} & -2a_{3z} & 0 & 2a_{3x} & -a_{3y} & 2a_{3x} & 0 & a_{3y} & 2a_{3z} & -a_{3y} \\ a_{3z} & 2a_{3y} & -2a_{3x} & 0 & -a_{3z} & 0 & 2a_{3x} & -a_{2z} & 2a_{3y} & a_{3z} \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} q_0^2 \\ q_0 q_1 \\ q_0 q_2 \\ q_0 q_3 \\ q_1^2 \\ q_1 q_2 \\ q_1 q_3 \\ q_2^2 \\ q_2 q_3 \\ q_3^2 \end{bmatrix} = \begin{bmatrix} b_1 x \\ b_1 y \\ b_1 z \\ b_2 x \\ b_2 y \\ b_2 z \\ b_3 x \\ b_3 y \\ b_3 z \\ 1 \end{bmatrix} \tag{H-5}$$

The 10×10 coefficient matrix can be inverted with Mathematica. And we find that the solution for the 10 products of the 4 quaternion components can be expressed in terms of scalar triple products as follows:

$$q_0^2 = \frac{\det(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3) + \det(\boldsymbol{b}_1, \boldsymbol{a}_2, \boldsymbol{a}_3) + \det(\boldsymbol{a}_1, \boldsymbol{b}_2, \boldsymbol{a}_3) + \det(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{b}_3)}{4\det(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3)} \tag{H-6}$$

$$q_1^2 = \frac{\det(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3) + \det(P_{11}\boldsymbol{b}_1, \boldsymbol{a}_2, \boldsymbol{a}_3) + \det(\boldsymbol{a}_1, P_{11}\boldsymbol{b}_2, \boldsymbol{a}_3) + \det(\boldsymbol{a}_1, \boldsymbol{a}_2, P_{11}\boldsymbol{b}_3)}{4\det(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3)} \tag{H-7}$$

$$q_2^2 = \frac{\det(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3) + \det(P_{22}\boldsymbol{b}_1, \boldsymbol{a}_2, \boldsymbol{a}_3) + \det(\boldsymbol{a}_1, P_{22}\boldsymbol{b}_2, \boldsymbol{a}_3) + \det(\boldsymbol{a}_1, \boldsymbol{a}_2, P_{22}\boldsymbol{b}_3)}{4\det(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3)} \tag{H-8}$$

$$q_3^2 = \frac{\det(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3) + \det(P_{33}\boldsymbol{b}_1, \boldsymbol{a}_2, \boldsymbol{a}_3) + \det(\boldsymbol{a}_1, P_{33}\boldsymbol{b}_2, \boldsymbol{a}_3) + \det(\boldsymbol{a}_1, \boldsymbol{a}_2, P_{33}\boldsymbol{b}_3)}{4\det(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3)} \tag{H-9}$$

$$q_i q_j = \frac{\det(P_{ij}\boldsymbol{b}_1, \boldsymbol{a}_2, \boldsymbol{a}_3) + \det(\boldsymbol{a}_1, P_{ij}\boldsymbol{b}_2, \boldsymbol{a}_3) + \det(\boldsymbol{a}_1, \boldsymbol{a}_2, P_{ij}\boldsymbol{b}_3)}{4\det(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3)} \tag{H-10}$$

[1]Micheals RJ, Boult TE. Increasing robustness in self-localization and pose estimation. [date unknown; accessed 2010 Jun]. http://www.vast.uccs.edu/ tboult/PAPERS/SPIE99-Increasing-robustness-in-self-localization-and-pose-estimation-Micheals-Boult.pdf.

where

$$\begin{aligned}
\det(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}) &= \det\begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix} \\
&= a_x(b_y c_z - b_z c_y) + a_y(b_z c_x - b_x c_z) + a_z(b_x c_y - b_y c_x) \\
&= \boldsymbol{a}\cdot(\boldsymbol{b}\times\boldsymbol{c}).
\end{aligned} \tag{H-11}$$

and

$$P_{11} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad P_{22} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \quad P_{33} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{H-12}$$

$$P_{01} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}, \quad P_{02} = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad P_{03} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{H-13}$$

$$P_{12} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad P_{13} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad P_{23} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{H-14}$$

We set q_0 as the positive square root of Eq. H-6 and then use Eq. H-10 to get $q_1 = q_0 q_1/q_0$, $q_2 = q_0 q_2/q_0$, and $q_3 = q_0 q_3/q_0$. The axis of rotation is along the unit vector

$$\hat{\mathbf{u}} = \frac{q_1\hat{\mathbf{\imath}} + q_2\hat{\mathbf{\jmath}} + q_3\hat{\mathbf{k}}}{\sqrt{1-q_0^2}}, \tag{H-15}$$

and the angle of rotation is

$$\theta = 2\cos^{-1} q_0. \tag{H-16}$$

The full rotation matrix is

$$R = \begin{bmatrix} 2q_0^2 - 1 + 2q_1^2 & 2q_1q_2 - 2q_0q_3 & 2q_1q_3 + 2q_0q_2 \\ 2q_1q_2 + 2q_0q_3 & 2q_0^2 - 1 + 2q_2^2 & 2q_2q_3 - 2q_0q_1 \\ 2q_1q_3 - 2q_0q_2 & 2q_2q_3 + 2q_0q_1 & 2q_0^2 - 1 + 2q_3^2 \end{bmatrix}. \tag{H-17}$$

The program in Listing H-1 is designed to test the implemented closed-form solution.

Listing H-1. ao.cpp

```
// ao.cpp: test of absolute orientation as implemented in Rotation class

#include "Rotation.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>
using namespace va;   // vector algebra namespace

int main( int argc, char* argv[] ) {

   Vector a1( 3.5, 1.0, 2.3 ), a2( 1.5, 2.1, 7.1 ), a3( 4.3, -5.8, 1.7 );   // 3 linearly independent vectors
   Vector b1, b2, b3;

   double pitch = rad( 45. );    // default pitch (deg converted to radians)
   double yaw   = rad( -30. );   // default yaw (deg converted to radians)
   double roll  = rad( 60. );    // default roll (deg converted to radians)
   if ( argc == 4 ) {   // or specify pitch, yaw, roll (deg) on command line

      pitch = rad( atof( argv[1] ) );
      yaw   = rad( atof( argv[2] ) );
      roll  = rad( atof( argv[3] ) );
   }

   std::cout << std::setprecision(6) << std::fixed << std::showpos;
   std::cout << "The 3 vectors are linearly independent iff det(a1,a2,a3) is non-zero: ";
   std::cout << "det(a1,a2,a3) = " << ( a1 * ( a2 ^ a3 ) ) << std::endl << std::endl;

   // perform rotation sequence on initial vectors
   Rotation R1( pitch, yaw, roll, XYZ );
   b1 = R1 * a1;
   b2 = R1 * a2;
   b3 = R1 * a3;

   // output the rotated vectors
   std::cout << "The rotated vectors are:" << std::endl;
   std::cout << "b1 = " << b1 << std::endl;
   std::cout << "b2 = " << b2 << std::endl;
   std::cout << "b3 = " << b3 << std::endl << std::endl;

   // given only the two sets of vectors, find the rotation that takes {a1,a2,a3} to {b1,b2,b3}
   Rotation R2( a1, a2, a3, b1, b2, b3 );

   // apply this rotation to the original vectors
   b1 = R2 * a1;
   b2 = R2 * a2;
   b3 = R2 * a3;

   // output rotated vectors to show they match previous output
   std::cout << "The following vectors should match those above:" << std::endl;
   std::cout << "b1 = " << b1 << std::endl;
   std::cout << "b2 = " << b2 << std::endl;
   std::cout << "b3 = " << b3 << std::endl << std::endl;

   // factor this rotation into a pitch-yaw-roll rotation sequence
   sequence s = factor( R2, XYZ );

   // output rotation sequence to show it matches the input values
   std::cout << "Factoring this rotation into a rotation sequence gives:" << std::endl;
   std::cout << "pitch = " << deg( s.first ) << std::endl;
   std::cout << "yaw   = " << deg( s.second ) << std::endl;
   std::cout << "roll  = " << deg( s.third ) << std::endl;

   return EXIT_SUCCESS;
}
```

Compiling this program with

```
g++ -O2 -Wall -o ao ao.cpp -lm
```

and then running it with the command

```
./ao -35.2 43.5 -75.6
```

prints out the following:

```
1  The 3 vectors are linearly independent iff det(a1,a2,a3) is non-zero: det(a1,a2,a3) = +143.826000
2
3  The rotated vectors are:
4  b1 = +2.917177 -2.334937 +2.139660
5  b2 = +6.633337 +1.253165 +3.390932
6  b3 = -2.129101 -2.066399 +6.798303
7
8  The following vectors should match those above:
9  b1 = +2.917177 -2.334937 +2.139660
10 b2 = +6.633337 +1.253165 +3.390932
11 b3 = -2.129101 -2.066399 +6.798303
12
13 Factoring this rotation into a rotation sequence gives:
14 pitch = -35.200000
15 yaw   = +43.500000
16 roll  = -75.600000
```

The 3 vectors need not be orthonormal—nor even mutually orthogonal—but they must be linearly independent. A necessary and sufficient condition for this is $\det(\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3) \neq 0$. We see that is the case from line 1. Lines 4–6 show the effect of the given rotation sequence upon the original 3 vectors (see lines 28–38 of Listing H-1). The program then computes the rotation that will take the original vectors to these 3 vectors (line 41 of Listing H-1) and then applies it to the original vectors (lines 43–52 of Listing H-1). We see on lines 9–11 that these do indeed match lines 4–6. Finally, the program factors the computed rotation into a pitch-yaw-roll rotation sequence (line 55 of Listing H-1) and the lines 14–16 show that we retrieve the input values. Thus we verify that the program is able to find the rotation as long as the original vectors are linearly independent.

INTENTIONALLY LEFT BLANK.

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO L IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
	<u>ABERDEEN PROVING GROUND</u>
12 (11 PDF, 1 HC)	RDRL SLB D J COLLINS RDRL SLB G J ABELL D CARABETTA T MALLORY RDRL SLB S J AUTEN R DIBELKA C HUNT J HUNT S MORRISON R SAUCIER (1 HC) G SAUERBORN

INTENTIONALLY LEFT BLANK.